Diplomarbeit

Im Studiengang Medieninformatik

MDA auf Basis Open Source

Abwicklung eines Pilotprojektes für einen Geschäftsvorfall einer Versicherung mit MDA auf Basis von Open Source

erstellt für

NovaTec
Ingenieure für neue Informationstechnologien GmbH

Vorgelegt von:
Andreas Keefer
an der Hochschule der Medien
am 25. Januar 2007

Erstprüfer:
Prof. Walter Kriha
Zweitprüfer:
Dipl. Math. (FH) Frank Gröhler

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Andreas Keefer
Stuttgart, den 25. Januar 2007

DANKSAGUNG

Ich danke Andreas Springer und Dorothea Klitsche für ihre Hilfe beim Korrekturlesen der Arbeit.

Natürlich danke ich auch Professor Walter Kriha und meinem Betreuer aus der Firma, Frank Gröhler, für die Betreuung und Unterstützung während der Diplomarbeit.

Kurzfassung

Diese Diplomarbeit befasst sich mit der Model Driven Architecture (MDA) und deren Anwendung auf Basis von Open Source.

Für ein Unternehmen ist es immer wichtig zu wissen, wie der aktuelle Stand der Technik ist und ob diese Technik produktiv eingesetzt werden kann, um sich dadurch Vorteile am Markt zu verschaffen.

Es werden zunächst einige Grundlagen von Open Source und MDA erörtert. Danach wird mittels einer Evaluierung von Open Source Modellierungs- und Generierungswerkzeugen ein Überblick über die am Markt verfügbaren Werkzeuge gegeben. Anschließend wird die prototypenhafte Umsetzung eines Geschäftsvorfalles einer Versicherung beschrieben, um den Stand der Technik zu demonstrieren. In der Schlussbetrachtung folgt ein Fazit über den Verlauf und die Resultate der Diplomarbeit. Zum Schluss wird ein Ausblick für die Model Driven Architecture gegeben.

Abstract

This diploma thesis is concerned with the Model Driven Architecture (MDA) and their appliance to basis of Open Source.

For an enterprise it is always important to know, how the current state of the art is and whether this technology can be used productively, in order to provide thereby advantages at the market.

First some bases of Open Source and MDA are discussed. Afterwards, by an evaluation of Open Source modelling and generation tools, an overview of the market of the supporting tools is given. Subsequently, the prototypeful implementation of a business use case of an insurance company is described, in order to demonstrate the state of the art. In the final consideration a result follows over the process and the results of the diploma thesis. In the end a view in the future for the Model Driven Architecture is given.

Inhaltsverzeichnis

Seite

1 Einleitung

Da heutige Software immer komplexer und der Konkurrenzdruck für die Gewinnung von Projekten immer schwieriger wird, muss ein Unternehmen versuchen, sich vom Markt abzuheben. Die Model Driven Architecture (MDA) bietet hierfür verschiedene Ansatzpunkte, welche in dieser Arbeit betrachtet werden.

1.1 Problemstellung

Die Technik entwickelt sich rasant weiter, deshalb ist es für ein Unternehmen wichtig zu wissen, wie der aktuelle Stand der Technik ist und ob diese Technik produktiv eingesetzt werden kann. Deshalb soll in diesem Fall durch eine Evaluierung von Open-Source-Werkzeugen der aktuelle Stand der unterstützenden MDA-Werkzeuge aufgezeigt werden.

Die Beschränkung auf Open Source, zum einen aus wirtschaftlichen Gründen und zum anderen um die Arbeit in einem gewissen Rahmen zu halten.

Ein Prototyp soll dann die tatsächliche Machbarkeit bzw. den aktuellen Stand der Technik demonstrieren.

1.2 Zielsetzung

Die Diplomarbeit soll eine Übersicht über die bestehenden Open-Source-Modellierungs- und Generierungswerkzeuge geben und dadurch den aktuellen Stand der MDA-Werkzeuge aufzeigen.

Anhand eines Prototypen sollen die verfügbaren Möglichkeiten und der Stand der Technik durch die Auswahl eines oder mehrerer evaluierter Werkzeuge aufgezeigt

werden. Es soll ebenfalls herausgearbeitet werden, wie weit die Modellierung einer Anwendung einen Sinn ergibt und ab wann der traditionelle Weg über manuelles Codieren vorteilhafter ist.

1.3 Abgrenzung

In der Arbeit werden ausschließlich Open-Source-Werkzeuge betrachtet und evaluiert. Ebenso habe ich mich auf den MDA-Aspekt konzentriert und versucht, nicht in die allgemeine modellgetriebene Entwicklung abzurutschen. Einige Übergriffe waren allerdings unvermeidlich, um Parallelen oder Unterschiede anzudeuten.

1.4 Vorgehen

Das Vorgehen während der Diplomarbeit kann grob in fünf Phasen eingeteilt werden:

- Die Einarbeitung in MDA und UML
- Die Evaluierung der Modellierungswerkzeuge
- Die Evaluierung der Generierungswerkzeuge
- Die Entwicklung des Prototypen
- Die Finalisierung der Diplomarbeit

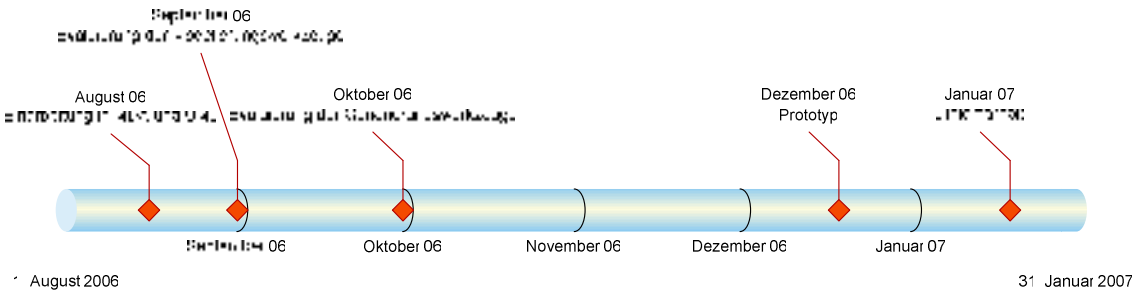

Abbildung 1-1: Zeitplan der Diplomarbeit mit den Phasen Meilensteinen

Mein Vorgehen während der praktischen Aufgaben war meistens so, dass ich parallel eine grobe Fassung in die Diplomarbeit dokumentiert habe. Dies hat die großen Vorteile, dass die Diplomarbeit parallel zu den praktischen Teilen entsteht, frühzeitig Probleme oder Engpässe erkannt werden können, am Ende weniger Hektik herrscht und Nichts vergessen wird.

1.5 Aufbau der Diplomarbeit

Die Arbeit ist grob in fünf Teile aufgeteilt:

- **Einführung:** Das erste Kapitel gibt eine Einführung in die Thematik und beschreibt die Fragestellung und Zielsetzung der Arbeit.
- **Theorieteil:** Die Kapitel 2 und 3 behandeln zum einen kurz das Thema Open Source und zum anderen den Bereich MDA.
- **Evaluierung:** Das 4. Kapitel beinhaltet die Evaluierung der MDA-Werkzeuge. Es werden zum einen Modellierungswerkzeuge und zum anderen Generierungswerkzeuge evaluiert.
- **Prototyp:** Das 5. Kapitel beschäftigt sich mit dem Prototypen, von der Analyse über das Design bis zur Implementierung der Anwendung.
- **Zusammenfassung:** Das Letzte, 6. Kapitel ist eine Zusammenfassung der Arbeit mit den Problemen und Erkenntnissen, die sich durch die Diplomarbeit ergeben haben.

2 Open Source

In diesem Kapitel wird der Begriff „Open Source“ näher erläutert und es wird näher darauf eingegangen, warum dieses Thema momentan so interessant ist. Es werden die wichtigsten Open Source Lizenzen vorgestellt, um einen Überblick zu erhalten und zu erkennen welche Besonderheiten sich aus den einzelnen Lizenzen ergeben.

2.1 Was ist Open Source?

Der Ausdruck „Open Source“ wird meist mit Computer-Software assoziiert und meint im Sinne der Open-Source-Definition, dass es jedem ermöglicht wird Einblick in den Quelltext eines Programms zu haben, sowie die Erlaubnis zu haben diesen Quellcode auch beliebig weiterzugeben oder zu verändern (vgl. [Wikipedia2006a]).

2.2 Was macht Open Source so interessant?

Im privaten Bereich dürfte das sehr eindeutig sein. Die Open-Source-Software ist kostenlos und bietet deshalb natürlich das beste Preis-Leistungsverhältnis.

Im Unternehmensbereich spielen dabei aber noch einige andere Aspekte eine Rolle:

- Es fallen natürlich keine Lizenz- oder Anschaffungskosten an, wodurch die Software insgesamt einen Kostenvorteil gegenüber kommerziellen Produkten haben kann.
- Der Quellcode kann geprüft werden. Besonders bei sicherheitskritischen Anwendungen kann dieses Argument sehr überzeugend sein. Meist werden auch Sicherheitslöcher sehr viel schneller behoben als bei kommerzieller Software.

- Die Anwendung kann einfacher, der zugrunde liegenden Lizenz entsprechend, erweitert oder in bestehende Anwendungen integriert werden.
- Herstellerunabhängigkeit: Man ist nicht von einem einzigen Hersteller abhängig und kann die Software selber weiterentwickeln.
- Die Qualität der Software ist meist mit kommerziellen Produkten gleichzusetzen.
- Interoperabilität: Bei Open Source wird meist auf offene und anerkannte Standards gesetzt, dies erleichtert das Zusammenspiel mit anderen Anwendungen.

2.3 Open Source Lizenzen

Es gibt drei charakteristische Merkmale, die eine Lizenz einer Open-Source-Software erfüllen muss:

- Der Programmcode der Software liegt in einer für Menschen lesbaren und verständlichen Form vor.
- Die Software darf beliebig kopiert, verbreitet und genutzt werden.
- Die Software darf verändert und in der veränderten Form weitergegeben werden.

Das Copyleft-Prinzip

Eine veränderte Software darf nur dann verbreitet werden, wenn sie selbst wieder unter derselben Lizenz lizenziert wird. Ziel dieses Prinzips ist es, die Freiheiten einer Software in der Weiterentwicklung von Anderen zu sichern, nach dem Wahlspruch: „Was frei ist soll frei bleiben!“.

Das Copyleft-Prinzip findet sich beispielsweise in den GNU-Lizenzen oder unter dem Begriff „Share Alike“ in einigen Creative Commons-Lizenzen.

2.3.1 GNU General Public License (GPL)

Die GPL gewährt im Grunde folgende Freiheiten:

- Die Software darf ohne jede Einschränkung für jeden Zweck genutzt werden, auch Kommerziell.
- Kopien dürfen kostenlos oder gegen Geld verteilt werden, wobei der Programmcode mitverteilt werden oder zugänglich sein muss. Der Empfänger erhält dieselben Freiheiten. Lizenzgebühren sind nicht erlaubt. Niemand ist verpflichtet Kopien zu verteilen, wenn er es aber tut, dann nur nach den Regeln der GPL.

- Die Software darf verändert werden. Wird die veränderte Software weitergegeben, gelten die Regeln der GPL. Hier gilt auch das Copyleft-Prinzip.

2.3.2 GNU Lesser General Public License (LGPL)

Die LGPL gewährt im Grunde dieselben Freiheiten wie die GPL. Allerdings dürfen externe Programme, die eine mit der LGPL Lizenzierte Software nutzen, ihre eigene Lizenz behalten. Deshalb ist die LGPL besonders für Bibliotheken geeignet. Viele Standardbibliotheken von Programmiersprachen, beispielsweise die glibc-Bibliothek, sind unter der LGPL lizenziert.

2.3.3 Eclipse Public License (EPL)

Die EPL ist eine Abwandlung der Common Public License (CPL) und speziell auf die Eclipse Foundation abgestimmt. Die grundlegenden Freiheiten und Regeln sind jedoch dieselben. Anders als bei der GPL muss nicht jedes auf der Software basierende Programm auch unter die EPL gestellt werden. Kommt ein neuer Teil zu der Software hinzu, so muss dieser nicht unter die EPL gestellt werden. Wenn allerdings ein unter der EPL stehender Teil verändert wird, so muss dieser auch weiterhin unter der EPL vertrieben werden.

2.3.4 Berkeley Software Distribution (BSD) Lizenz

Das BSD Lizenzmodell ähnelt im Groben auch der GPL, allerdings enthält die Lizenz kein Copyleft. Wird eine unter der BSD-Lizenz stehende Software verändert, so muss der Programmcode der veränderten Software nicht veröffentlicht werden. Allerdings muss der Copyright-Vermerk der ursprünglichen Software enthalten bleiben. Software unter der BSD-Lizenz eignet sich somit gut als Grundlage für Kommerzielle Produkte.

3 Einführung in die MDA

Die Model Driven Architecture (MDA) versucht die schon lange existierende Idee, von der Trennung von fachlichen Aspekten und Technik umzusetzen (vgl. [OMG2003], 2.1.2 Overview). MDA erreicht dies im Grunde mit Hilfe von architektonischer Trennung der Zuständigkeiten über Modelle.

3.1 Allgemeines

MDA ist eine Strategie bzw. ein Standardisierungskonzept der Object Management Group (OMG). Die OMG ist eine offene Gesellschaft und erstellt herstellerunabhängige Spezifikationen zur Verbesserung der Interoperabilität und Portabilität von Enterprise-Anwendungen.

Da im klassischen Vorgehen mit (UML)-Modellen meist nur ein Modell existiert, in dem fachliche und technische Informationen vermischt sind, teilt MDA das Gesamtmodell in mehrere Schichten:

- **Computation Independent Model (CIM).** Das CIM beschreibt ein System aus der Sicht des Business und ist somit von der IT völlig unabhängig. Das CIM wird oft auch als „Domänen-Modell“ oder „Business-Modell“ bezeichnet.
- **Platform Independent Model (PIM).** Das PIM bietet eine plattformunabhängige Sicht auf ein System. Es kann somit für viele verschiedene Plattformen angewendet werden. Hier werden meist die Geschäftsprozesse abgebildet.
- **Platform Specific Model (PSM).** Das PSM verbindet die unabhängigen Spezifikationen des PIM mit den plattformspezifischen Gegebenheiten der entsprechenden Plattform.
- **Zielplattform (Code).** Aus dem PSM wird der Quellcode für die Zielplattform generiert. Meist entsteht dabei noch kein ausführbarer Code, sondern lediglich ein Grundgerüst, das für die weitere händische Implementierung genutzt wird.

Allerdings konnte sich nur in einzelnen Nischenmärkten, beispielsweise im Bereich Echtzeitsysteme, in dem schon immer eine hohe Präzision der Modelle gefordert wurde, die vollständige Modellierung durchsetzen (vgl. [Hitz2005], S.348).

Durch diese Schichten wird die Trennung der Zuständigkeiten (separation of concerns) erreicht. Durch die abstrakten Modelle will die MDA nicht nur eine Plattformunabhängigkeit erreichen, sondern auch eine Sprach- und Systemunabhängigkeit.

Mit Hilfe von Transformationen können die Modelle in anderen Schichten abgebildet werden. MDA unterscheidet dabei zwei Typen von Transformationen:

- **Modell-zu-Modell Transformation.** Ein Modell wird in ein anderes Modell transformiert. Somit kann z.B. ein PIM mit Hilfe von Transformationsregeln in ein PSM transformiert werden.
- **Modell-zu-Code Transformation.** Ein Modell wird in den Source Code einer bestimmten Programmiersprache transformiert. Diese Methode wird schon längere Zeit genutzt, z.B. in CASE-Werkzeugen, um Code aus Modellen zu generieren.

Die Transformationen erzeugen dabei aus den Elementen des Quellmodells die Elemente des Zielmodells. Üblicherweise laufen die Transformationen von der abstrakteren zur konkreteren Schicht (CIM ► PIM ► PSM ► Code).

Da MDA keinen festen Standard darstellt, sondern eher eine Strategie, ist die MDA auch sehr offen und bietet viel Spielraum für Interpretationen. Die OMG sieht für die Umsetzung von MDA die Verwendung von UML-Modellen vor. Generell ist es aber auch denkbar MDA mit einer andere Modellierungssprache umzusetzen, wenn diese sich an die Spezifikation der MDA hält (siehe Abschnitt 3.3).

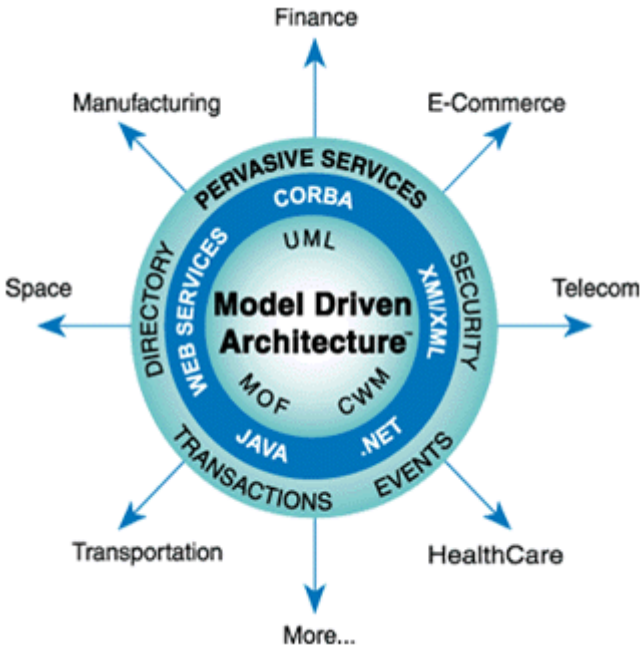

Abbildung 3-1: MDA Strategie (vgl. [OMG2006])

MDA kann mit dem damaligen Evolutionsschritt von der Assembler-Entwicklung zu den kompilierbaren Hochsprachen verglichen werden. Die Model Driven Architecture macht dem Schritt von Code basierten Hochsprachen zu abstrakteren, plattformunabhängigen Modellen.

3.2 Ziele der MDA

- **Interoperabilität.** Durch saubere, offene und anerkannte Standards wie UML (Unified Modeling Language), MOF (Meta-Object Facility) oder XMI (XML Metadata Interchange) soll die Interoperabilität von verschiedenen Werkzeugen und Systemen vereinfacht werden.
- **Plattform Unabhängigkeit.** Die Realisierung, also, wie die Software erstellt werden soll, ist sehr wechselhaft. Vor nicht all zu ferner Zeit waren es noch COBOL, C++ oder CORBA, heute sind es Technologien wie WebServices, XML oder EJBs und in Zukunft wird es immer wieder neue Technologien und Plattformen geben. MDA versucht, dies mit Hilfe der verschiedenen Abstraktionsstufen der Modelle zu lösen.
- **Höhere Wiederverwendbarkeit und Langlebigkeit.** Durch die verschiedenen Abstraktionsstufen der Modelle sind diese jeweils von der darunter liegenden Schicht entkoppelt und können so leichter wieder verwendet werden.
- **Konsistenz zwischen Code und Modellen.** In vielen Unternehmen werden bereits Modelle in der Analyse und im Design eingesetzt. Oft werden diese Modelle aber bei Änderungen nicht angepasst und somit im weiteren Projektablauf unbrauchbar. Daraus resultiert, dass die Modelle auch selten bei der Implementierung genutzt werden können und immer wieder von vorn begonnen werden muss. Ein Grund hierfür ist sicherlich auch die mangelnde Tool Unterstützung, bzw. Inkompatibilität zwischen den Tools wegen proprietärer Schnittstellen. MDA liefert hierfür eine solide Grundlage mit offenen Standards wie UML, MOF, XMI oder OCL (Object Constraint Language). Somit kann durch eine bessere Tool-Pipeline die Konsistenz zwischen Code und Modellen eher gewährleistet werden.
- **Steigerung der Entwicklungsgeschwindigkeit.** Dadurch, dass die Modelle im Entwicklungsprozess durchgehend verwendet werden können und am Ende Code generiert werden kann, verkürzt sich die Entwicklungszeit. Vor allem in zukünftigen Projekten, wenn Modelle oder Transformationen wieder verwendet werden können.
- **Verbesserung der Softwarequalität.** Wenn im Team entwickelt wird, entsteht bei jeder beteiligten Person qualitativ unterschiedlicher Code. Durch die

Transformation der Modelle kann eine konstante und reproduzierbare Qualität erreicht werden. Ebenso ist es möglich Tests oder Simulationen auf die Modelle anzuwenden.

- **Bessere Dokumentation.** Das Modell ist einfacher lesbar sowohl von Entwicklern als auch vom Fachbereich. Die Dokumentation kann mit hoher Qualität automatisch aus den Modellen abgeleitet werden, somit steigt auch die Produktivität.

3.3 Zusammenhang von MDA mit MOF, UML, XMI und QVT

Die Meta Object Facility (MOF) ist ein Meta-Metamodell zur Definition von Metamodellen. Andere Metamodelle, wie das der Unified Modeling Language (UML) oder des Common Warehouse Metamodel (CWM) und auch MOF selbst, wurden auf Basis von MOF definiert. MOF ist aber nicht nur zur Definition von Metamodellen geeignet, sondern bietet auch Funktionalität um Metadaten zu manipulieren. Es können Elemente von Metamodellen erzeugt, abgefragt, manipuliert und gelöscht werden. MOF bietet dabei auch Interoperabilität zwischen Metamodellen (PIMs), die auf Basis von MOF definiert wurden. Zum Beispiel erlaubt es der XMI-Standard, MOF-Metamodelle auf XML-Schemadefinitionen in Form von DTDs oder XML-Schema abzubilden (vgl. [Hitz2005], S.328).

Historisch gesehen entstand MOF aus dem Sprachkern von UML 1, allerdings bestanden noch einige essentielle Unterschiede. Eine vollständige Integration und damit auch eine vollständige Nutzung der Vorteile beider Standards wurde dadurch verhindert. Aus diesem Grund wurde bei der Entwicklung von UML 2 besonders darauf geachtet, eine architektonische Abstimmung beider Standards zu erreichen. Es wurde eine gemeinsame Basis in Form des *Core*-Paketes der *Infrastructure-Library* geschaffen. Dieser Kern wird sowohl von Meta-Metamodellen wie MOF (auf M3) als auch von anderen Metamodellen, wie dem UML-Metamodell, auf M2 verwendet (vgl. [Hitz2005], S.328). Das heißt, dass das UML 2 Metamodell auch eine Instanz des MOF Meta-Metamodells ist, und somit alle Vorteile von MOF nutzen kann.

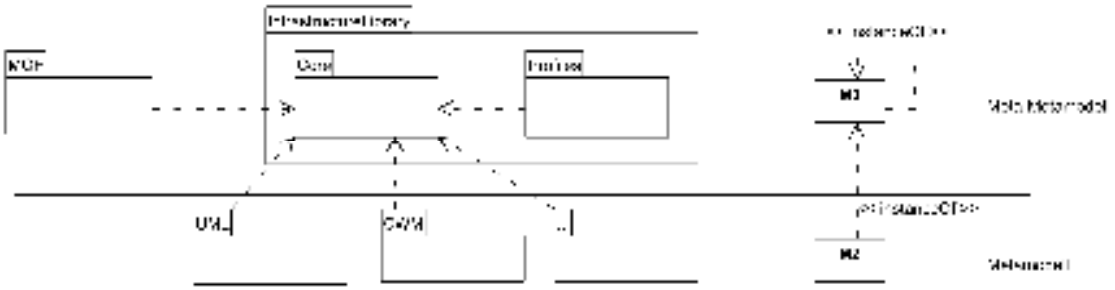

Abbildung 3-2: Zusammenhang zwischen Infrastruktur, MOF, UML und anderen Metamodellen (vgl. [Hitz2005], S. 329)

Somit ist jetzt ein plattformunabhängiges System zum Austausch von Modellen entstanden. Es kann z.B. Ein UML 2 Modell mittels XMI auf eine andere Modellierungssprache, die auf MOF basiert, wie CWM, abgebildet werden. Für diese Abbildung ist allerdings eine Metamodell Transformation notwendig.

Für (Meta)modell Transformationen empfiehlt die OMG die Verwendung von MOF QVT. Das Akronym QVT steht für "queries" (Anfragen), "views" (Sichten) und "transformations" (Transformationen).

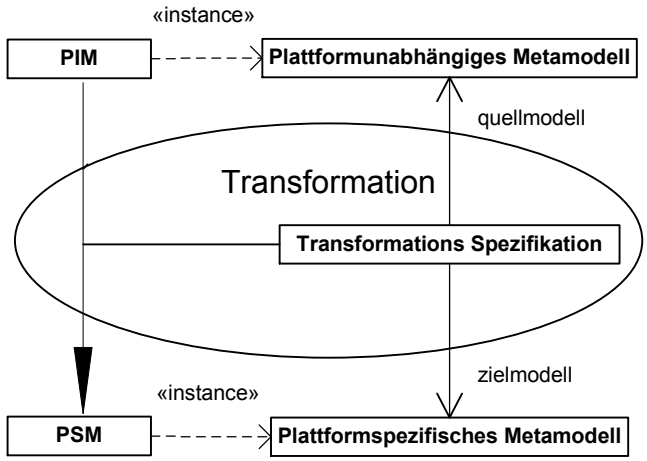

Abbildung 3-3: Metamodell Transformation (vgl. [OMG2003], Figure 3-2)

Daraus folgt, dass jeder Werkzeughersteller sein eigenes (proprietäres) MOF-basiertes Metamodell verwenden könnte, aber trotzdem ein Austausch der Modelle gewährleistet ist.

3.4 Konzepte

In diesem Teil der Arbeit werden die Kernkonzepte der MDA kurz erläutert. Das sind die Modelle, die Plattform und die UML-Profile.

3.4.1 Modelle

Modelle sind eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten eines Systems (vgl. [Wikipedia2006]). Sie werden üblicherweise mit der Modellierungssprache UML definiert.

MDA-Modelle unterscheiden sich von gewöhnlichen UML-Modellen darin, dass ihre Bedeutung formal definiert ist. Dies wird durch UML-Profile erreicht (siehe unten). Es ist also nicht jedes UML-Modell auch gleichzeitig ein MDA-Modell und umgekehrt.

3.4.2 Plattform

Eine Plattform, und deren Abstraktionsgrad, ist durch die MDA nicht fest definiert. Eine Plattform kann dann beispielsweise die X86 Architektur für ein Windows Betriebssystem oder J2EE für eine Webanwendung sein. Da die OMG aber vor allem an der Verbesserung der Interoperabilität und Portabilität von Enterprise-Anwendungen interessiert ist, spiegelt sich das auch in der praktischen Bedeutung des Begriffes wieder.

3.4.3 UML-Profile

UML-Profile stellen die Möglichkeit zur Erweiterung des UML-Metamodells dar. Sie erlauben die Formalisierung von Domänenmodellen. Das bedeutet, dass mittels UML-Profilen der UML-Sprachumfang so erweitert werden kann, dass domänen- oder plattformspezifische Semantiken abgebildet werden können, beispielsweise EJBs.

3.5 Abgrenzung

MDA wird gelegentlich mit anderen Technologien oder Konzepten gleichgesetzt:

- MDA wir oft synonym zu den CASE-Ansätzen verwendet. MDA und CASE unterscheiden sich jedoch in ihren Zielen. Die CASE-Ansätze haben zum Ziel, die Softwareentwicklung durch den Einsatz von IT-gestützten Werkzeugen zu unterstützen. Fachliche Anforderungen werden dabei möglichst vollständig und automatisiert erstellt. Somit ist eine Trennung von fachlichen und technischen Aspekten nicht gegeben. Meist sind Metamodelle und Transformationen vorgegeben und lassen sich nicht anpassen. Oft werden proprietäre Modellierungssprachen zur Spezifikation der Systeme verwendet, was das Mindestmaß an Interoperabilität verletzt.

- Oft wird MDA auch mit der Generierung gleichgesetzt. Dem ist aber nicht so. Die Generierung von Code ist nur ein Teil, der MDA ausmacht. MDA abstrahiert hier auch einen Schritt weiter. So spricht man im Kontext von MDA eher von Transformation und nicht von Generierung, da MDA nicht nur Modell-zu-Code Transformationen vorsieht, sondern auch Modell-zu-Modell Transformationen.
- Zu beachten ist auch, dass MDA nicht mit der Modellgetriebenen Softwareentwicklung (MDSD) gleichzusetzen ist. MDA ist dabei nur eine mögliche Umsetzung von MDSD. MDSD ermöglicht es, Softwaresysteme durchgängig mit Modellen zu beschreiben. Das Modell ist dabei der Ausgangspunkt für den Entwicklungsprozess. MDA spezifiziert diese allgemeine Aussage detaillierter, beispielsweise durch plattformunabhängige und plattformspezifische Modelle.

4 Evaluierung der MDA Werkzeuge

In diesem Kapitel werden verschiedene Open Source Werkzeuge, die momentan am Markt erhältlich sind, vorgestellt, auf ihre Stärken und Schwächen hin untersucht und miteinander verglichen. Am Ende wird ein oder auch mehrere Werkzeuge für die prototypenhafte Umsetzung des vorgegebenen Geschäftsvorfalles ausgewählt.

4.1 Übersicht

Es existiert eine große Anzahl an verschiedenen MDA-Werkzeugen, deren Funktionsumfang und Einsatzgebiet stark variieren. Dies ist nicht verwunderlich, da der MDA-Ansatz sehr offen und allgemein für die gesamte Softwareentwicklung definiert ist. Es können beispielsweise Webanwendungen, Echtzeitsysteme oder verteilte Anwendungen gleichermaßen auf Basis der MDA entwickelt werden (vgl. [Hitz2005], S. 350).

4.1.1 Abgrenzung

Im Moment existiert leider kein generisches MDA-Werkzeug, mit dem sämtliche Arten an Software abgedeckt werden können. Für eine bestimmte Art von Anwendung muss man ein entsprechendes Werkzeug aus der Masse der verfügbaren MDA-Werkzeuge auswählen. Diese Werkzeuge können grob in drei Werkzeugkategorien eingeteilt werden:

- **xUML-Werkzeuge.** Diese Werkzeugkategorie benutzt Executeable UML (xUML), dass eine Variante von UML darstellt und ausführbare Modelle unterstützt. Mit diesen Werkzeugen kann die komplette Anwendung modelliert werden und eine lauffähige Anwendung generiert werden. Diese Werkzeuge werden vorwiegend für Echtzeit- und Embeddedsysteme eingesetzt und produzieren meist C, C++ oder Ada Code (vgl. [Hitz2005], S. 350). Vertreter dieser Werkzeugkategorie ist beispielsweise iUML oder BridgePoint.

- **Plattformspezifische Werkzeuge.** Werkzeuge dieser Kategorie sind für eine spezielle Technologie ausgerichtet. Durch die Einschränkung auf eine bestimmte Plattform können diese Werkzeuge besondere Funktionalitäten bieten, wie beispielsweise eine automatische Verteilung von Komponenten. Leider ist der Benutzer aber auch durch diese Plattformabhängigkeit an die entsprechende Technologie gebunden. Speziell für objektorientierte Komponententechnologien wie J2EE oder .NET werden viele Werkzeuge angeboten (vgl. [Hitz2005], S. 350). OptimalJ ist beispielsweise ein Vertreter dieser Werkzeugkategorie.
- **Plattformunabhängige Werkzeuge.** Werkzeuge dieser Kategorie setzen einen großen Teil der MDA-Konzepte um. Sie unterstützen mehrere Zielplattformen und lassen sich an die Bedürfnisse der Benutzer gut anpassen. Werkzeuge wie AndroMDA können dieser Kategorie zugeordnet werden.

Die Grenze zwischen plattformspezifischen und plattformunabhängigen Werkzeugen ist sehr fließend und es ist deshalb oft sehr schwer bestimmte Werkzeuge genau einer Werkzeugkategorie zuzuordnen. Dies hängt auch vom Verständnis der Plattform ab. Ab wann ist ein Modell plattformunabhängig? Wenn es von der Programmiersprache unabhängig ist? Oder gar erst, wenn es ein reines Businessmodell ist? Diese Fragestellung muss jeder für sich selbst beantworten, da sie explizit von den Anforderungen der Software abhängt. Im Enterprise Umfeld ist allerdings meist die Unabhängigkeit von bestimmten Technologien wie J2EE, .Net oder WebServices gemeint.

Eine weitere Kategorie sind die reinen Codegeneratoren, die meist in CASE-Werkzeugen verwendet werden. Sie zählen nicht zu den MDA-Werkzeugen, da sie die MDA-Konzepte nicht beachten. Diese Werkzeuge können meist nur Code für eine vorgegebene Plattform erzeugen. Meist ist das eine Programmiersprache wie Java, C# oder PHP. Üblicherweise erzeugen sie aus Klassendiagrammen die entsprechenden Klassen der Programmiersprache mit Attributen und den Methodenrümpfen. Die Transformation lässt sich üblicherweise nicht oder nur sehr begrenzt anpassen. Diese Art von Werkzeugen wird deshalb für die Evaluierung ausgeschlossen.

Am Open Source Markt existieren momentan kaum komplette MDA-Werkzeuge. Fast immer sind diese in einen Modellierungs- und einen Generierungsteil aufgesplittet. Daher auch die Aufteilung der Evaluierung in Modellierungswerkzeuge und Generierungswerkzeuge.

4.1.2 Allgemeines Vorgehen für die Evaluierung

Als erstes werden die Anforderungen an das Werkzeug gesammelt, beschrieben und zu Anforderungskategorien zusammengefasst. Der nächste Schritt ist die Gewichtung der Anforderungen, da sie unterschiedliche Prioritäten haben. Die Gewichtung wird dabei in fünf Stufen eingeteilt.

Tabelle 4-1: Gewichtungen und ihre Bedeutung

Gewichtung	Bedeutung
1	~ unwichtige Anforderung
2	~ mäßig wichtige Anforderung
3	~ wichtige Anforderung
4	~ sehr wichtige Anforderung
5	~ unverzichtbare Anforderung

Als nächstes wurden die Werkzeuge nacheinander auf die einzelnen Anforderungen hin untersucht und die Umsetzung anschließend bewertet. Für die Bewertung stehen insgesamt sechs Stufen zur Verfügung.

Tabelle 4-2: Bewertungen und ihre Bedeutung

Bewertung	Bedeutung
0	~ nicht umgesetzt
1	~ sehr wenig umgesetzt
2	~ Grundlagen wurden umgesetzt
3	~ ausreichend umgesetzt
4	~ gut umgesetzt
5	~ voll umgesetzt

Gewichtet und bewertet werden die Anforderungen über die Anforderungskategorien um die Einfachheit und die Übersichtlichkeit zu gewährleisten.

Das Ergebnis der Evaluierung einer Anforderung errechnet sich durch eine Multiplikation der Gewichtung mit der Bewertung (Bewertung * Gewichtung = Anforderungsergebnis). Das Gesamtergebnis errechnet sich durch die Addition der einzelnen Anforderungsergebnisse (Anforderungsergebnis 1 + Anforderungsergebnis n = Gesamtergebnis). Somit erhält man für jedes Werkzeug eine Summe, mit deren Hilfe man die Umsetzung der Anforderungen schnell vergleichen kann.

Auf eine detaillierte Zuordnung von der Bedeutung einer Bewertung einer Anforderung wurde bewusst verzichtet. Dies hätte dem Evaluierungsprozess eine unnötige Starrheit gegeben und da die Anforderungen nicht einzeln, sondern über Anforderungskategorien gewichtet und bewertet werden, wäre so eine unnötig hohe Komplexität entstanden.

Tabelle 4-3: Beispiel für eine detaillierte Zuordnung von der Bedeutung einer Bewertung für die Anforderung *Interoperabilität* bei Modellierungswerkzeugen

Bewertung	Bedeutung	Anforderungsumsetzung
0	~ nicht umgesetzt	Kein Austausch von Modellen möglich.
1	~ sehr wenig umgesetzt	Austausch nur in proprietärem Format.
2	~ Grundlagen wurden umgesetzt	XMI in einer älteren Version unterstützt.
3	~ ausreichend umgesetzt	Einige XMI Versionen Unterstützt.
4	~ gut umgesetzt	Fast alle XMI Versionen für Import/Export unterstützt.
5	~ voll umgesetzt	Alle XMI Versionen für Import/Export unterstützt, zusätzlich Unterstützung für verbreitete proprietäre Formate.

4.2 Evaluierung von Modellierungswerkzeugen

Unter Modellierungswerkzeugen versteht man Werkzeuge, die in erster Linie zum Erstellen von Modellen benutzt werden. Die Modelle werden üblicherweise nach der Erstellung in einem bestimmten Format, beispielsweise XMI, abgespeichert und dann von einem Generierungswerkzeug wieder eingelesen.

Einige Modellierungswerkzeuge besitzen selbst auch einen integrierten Code-Generator, der aber an die Funktionalität und Flexibilität der Generierungswerkzeuge meist nicht heran reicht.

4.2.1 Einschränkungen

Da das Thema dieser Arbeit „MDA auf Basis Open Source“ ist, werden grundsätzlich nur Open-Source-Werkzeuge betrachtet. Die frei erhältlichen Community Versionen der kommerziellen Werkzeuge werden nicht untersucht.

Am Markt existieren viele Modellierungswerkzeuge, die sich jedoch in ihrer Qualität als auch ihrer Funktionalität sehr stark unterscheiden. Viele Werkzeuge basieren auf proprietären Metamodellen und bieten oft keine Möglichkeit zum standardisierten Import oder Export der Modelle, beispielsweise im XMI-Format. Einige beherrschen nicht ein Mindestmaß an Diagrammtypen oder sie sind unvollständig implementiert. Einige Werkzeuge sind auch noch in einem sehr frühen Entwicklungsstadium oder haben eine katastrophale Usability, weshalb sie nicht vernünftig zu bedienen sind.

Daher beschränkt sich die Evaluierung nur auf wenige Werkzeuge. Im Abschnitt 4.2.4 wird dennoch eine Übersicht über alle, zur Zeit der Evaluierung gefundenen Werkzeuge gegeben, auch wenn diese für die detaillierte Evaluierung ausgeschlossen wurden.

4.2.2 Anforderungen

Die Anforderungen an ein Modellierungswerkzeug im Bezug auf die Nutzung im MDA-Kontext:

- **Interoperabilität:** Das Modellierungswerkzeug sollte ein Mindestmaß an Interoperabilität bieten. Das heißt, das Werkzeug muss die erstellten Modelle exportieren können und Modelle anderer Werkzeuge möglichst importieren können. Ein gutes Austauschformat ist hier XMI, wobei es leider viele verschiedene zueinander inkompatible XMI-Versionen gibt.
- **Metamodell:** Die Modelle des Werkzeugs sollten auf einem MOF konformem Metamodell basieren, wie von MDA gefordert, beispielsweise UML. Dies erleichtert den Modellaustausch und auch die Modelltransformation.
- **Diagrammtypen:** Es muss mindestens das Klassendiagramm verfügbar sein. Um auch grafische Benutzeroberflächen modellieren zu können, sollte das Werkzeug auch Anwendungsfalldiagramme und Aktivitäts- oder Zustandsdiagramme beherrschen. Alle anderen Diagrammtypen sind meist nicht zwingend für die Weiterverarbeitung durch ein MDA-Generierungswerkzeuges nötig.
- **Wichtige (UML) Notationen:** Notationen der UML, wie Stereotypen, Eigenschaftswerte (Tagged Values) oder Einschränkungen (Constraints) werden oft von Generatoren verwendet, um entsprechende Ereignisse auszulösen oder die Codegenerierung zu beeinflussen. UML-Profile spielen dabei auch eine wichtige Rolle. Deshalb sind diese Aspekte auch entscheidend für die Auswahl des Modellierungswerkzeuges.
- **Teamunterstützung:** Heutzutage gibt es verschiedene Rollen in einem Softwareprojekt, die gemeinsam an den Modellen arbeiten müssen, beispielsweise Softwareanalysten, -architekten und -entwickler. Ohne Unterstützung des Werkzeuges ist es praktisch nicht möglich im Team an einem Modell gemeinsam zu arbeiten.
- **Support:** Eine aktive Community und eine gute Dokumentation ist für eine schnelle Lösung von Problemen bei einer Open Source Software sehr hilfreich. In Foren oder Newsgroups finden sich meist sehr schnell die dringend benötigten Antworten für spezielle Fragen. Aus diesem Grund ist dies auch eine wichtige Anforderung an ein Werkzeug.
- **Usability:** Das Werkzeug sollte sich intuitiv und möglichst einfach bedienen lassen. Dies reduziert die Einarbeitungsphase, spart Zeit bei der täglichen Bedienung und erhöht die Akzeptanz bei den Anwendern.
- **IDE Integration:** Idealerweise lässt sich das Werkzeug auch in die bestehende Entwicklungsumgebung integrieren, beispielsweise in Eclipse oder Microsofts Visual Studio.

4.2.2.1 Gewichtung

- **Interoperabilität (5):** Da die Interoperabilität bereits ein Grundbaustein der MDA ist, ist diese Anforderung unverzichtbar und wird deshalb mit fünf gewichtet.
- **Metamodell (4):** Auch das Metamodell wird von der MDA Spezifikation beschrieben. Es ist aber trotzdem nicht als unverzichtbar zu bewerten, da z.B. durch die Interoperabilität Schwächen beim Metamodell wieder kompensiert werden könnten. Deshalb die Gewichtung mit vier.
- **Diagrammtypen (4):** Die Vielfalt der Diagrammtypen ist sehr wichtig, da sie die Präzision und die Reichweite der Modelltransformation stark beeinflussen. Aus diesem Grund wird diese Anforderung ebenfalls mit vier gewichtet.
- **Wichtige (UML) Notationen (3):** Die Unterstützung der UML Notationen beeinflusst ebenfalls die Reichweite der Modelltransformation, allerdings nicht in dem Maße wie das die Diagrammtypen tun. Deshalb die Gewichtung mit drei.
- **Teamunterstützung (3):** Die Teamunterstützung ist ein wichtiger Punkt bei der Auswahl eines Werkzeuges. Je größer das Projekt, desto wichtiger ist die Mehrbenutzerunterstützung. Sie beeinflusst dabei aber nicht den MDA-Ansatz und wird deshalb mit drei gewichtet.
- **Support (3):** Durch einen guten Support kann bei Fragen und Problemen erheblich Zeit und dadurch Geld gespart werden. Allerdings hat diese Anforderung nicht so starken Einfluss auf den MDA-Ansatz und wird deshalb mit drei gewichtet.
- **Usability (3):** Durch eine gute Usability wird das Werkzeug schnell akzeptiert und es macht Spaß, damit zu arbeiten. Die Usability hat, ähnlich wie der Support, wenig Einfluss auf den MDA-Ansatz, deshalb auch hier die Gewichtung mit drei.
- **IDE Integration (1):** Durch die Integration in eine Entwicklungsumgebung verbessert sich auch die Usability und die Round-Trip-Zeiten können sich verkürzen. Da die IDE Integration aber auch als Unterpunkt der Usability gesehen werden kann, wird diese Anforderung sehr gering, mit eins gewichtet.

4.2.3 Vorgehen

In diesem Abschnitt möchte ich mein Vorgehen während der Evaluierung der Modellierungswerkzeuge beschreiben.

Zuerst habe ich die unten stehenden Quellen nach möglichen Werkzeugen durchsucht und in einer Tabelle aufgelistet.

Danach wurde die Tabelle um Informationen zu jenen Werkzeugen erweitert, die sich auf den jeweiligen Webseiten der Werkzeuge finden ließen. Diese Tabelle ist unten im Abschnitt 4.2.4 zu sehen. Hierbei sind einige Werkzeuge schon ausgeschieden, z.B. weil sie lediglich als Werkzeug zum Zeichnen von Diagrammen angepriesen wurden, ähnlich wie Microsoft Visio.

Quellen:

- http://oose.de/uml-tools
- http://www.objectsbydesign.com/tools/umltools_byPrice.html
- http://www.jeckle.de/umltools.htm
- http://google.de
- http://sourceforge.net

Als nächsten Schritt habe ich die restlichen Werkzeuge nacheinander installiert und ausprobiert. Dabei habe ich zuerst ein Klassendiagramm erstellt, das die wichtigsten Elemente dieses Diagrammtyps verwendet. Ebenso bin ich beim Anwendungsfall-, Aktivitäts- und Zustandsdiagramm vorgegangen. Die nachfolgenden Abbildungen zeigen die einzelnen Referenzdiagramme. Dabei ist darauf hinzuweisen, dass die Semantik der Diagramme irrelevant ist. Nur die technischen Funktionen und Elemente der Diagramme sind ausschlaggebend, beispielsweise, ob mehrfache Stereotypen möglich sind oder ob Signale (richtig) dargestellt werden.

Die Referenzdiagramme wurden mit der Community Edition von „poseidon“ und „MagicDraw“ erstellt. Es wurden genau diese Werkzeuge ausgewählt, weil sie von einigen Herstellern von Generierungswerkzeugen empfohlen werden. Damit existieren die Modelle im XMI 1.2 mit UML 1.4 Metamodell, XMI 2.1 mit UML 2.0 Metamodell und im EMF XMI 2.0 mit UML 2.0 Metamodell Format. Dadurch ist eine gute und breite Referenzgrundlage für die Evaluierung gegeben.

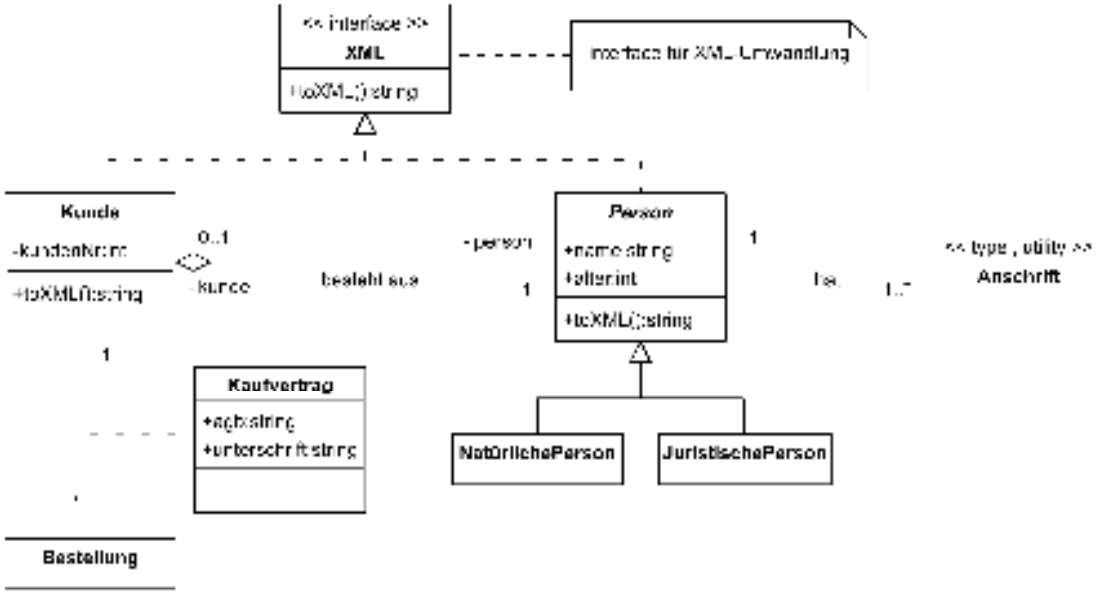

Abbildung 4-1: Referenz Klassendiagramm

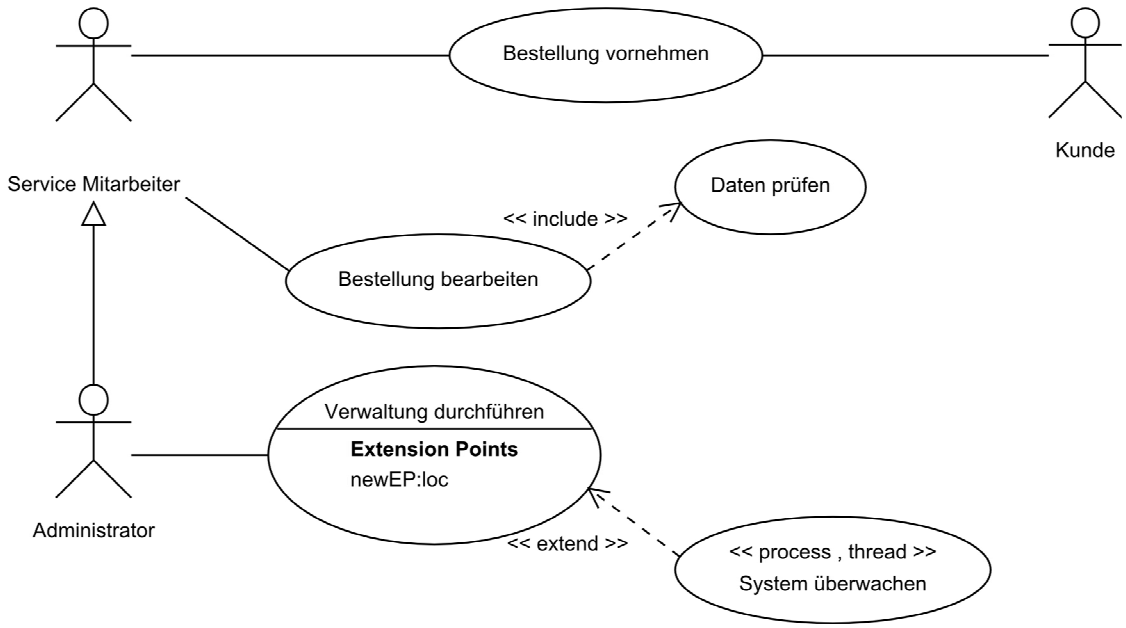

Abbildung 4-2: Referenz Anwendungsfalldiagramm

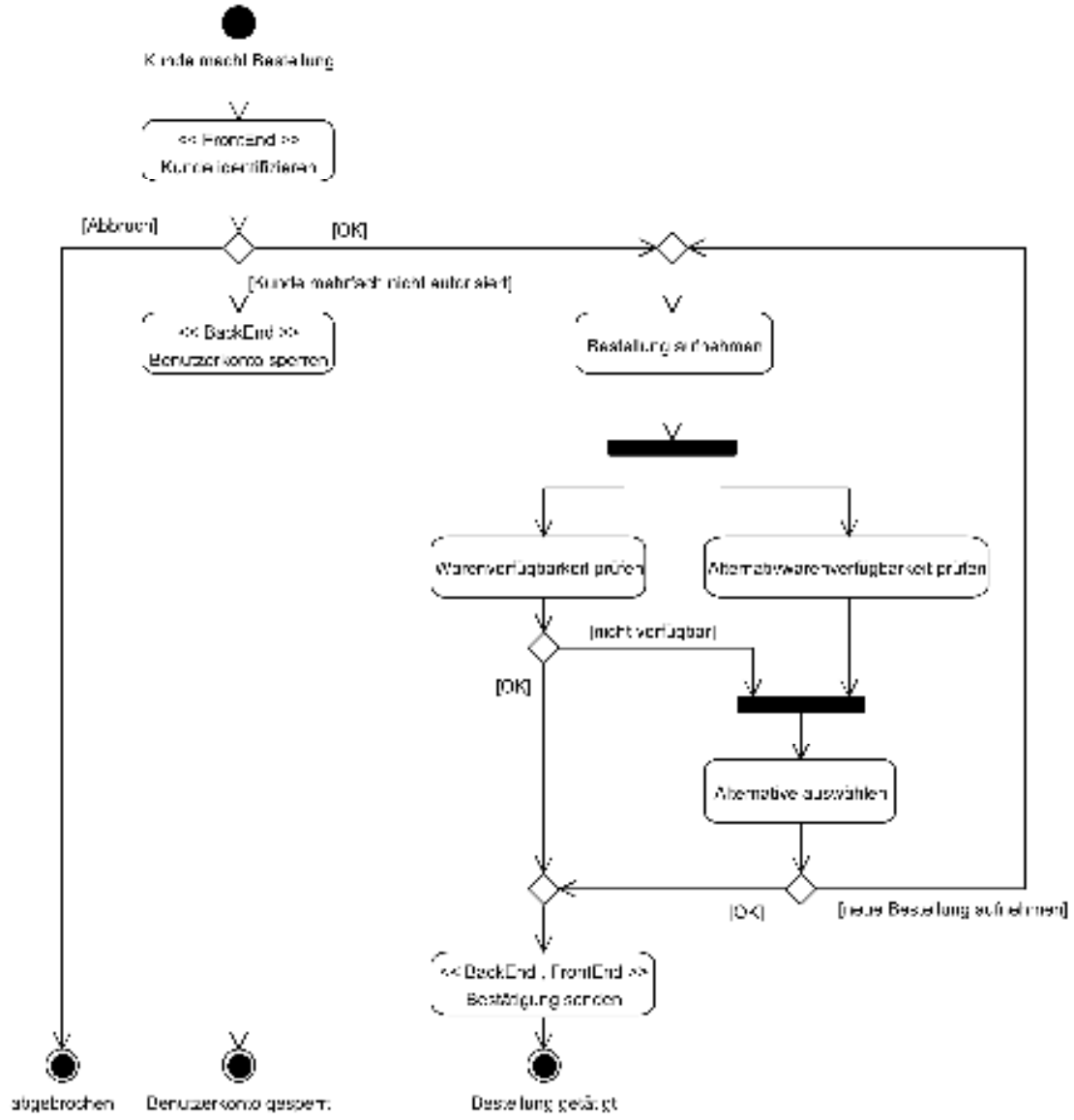

Abbildung 4-3: Referenz Aktivitätsdiagramm

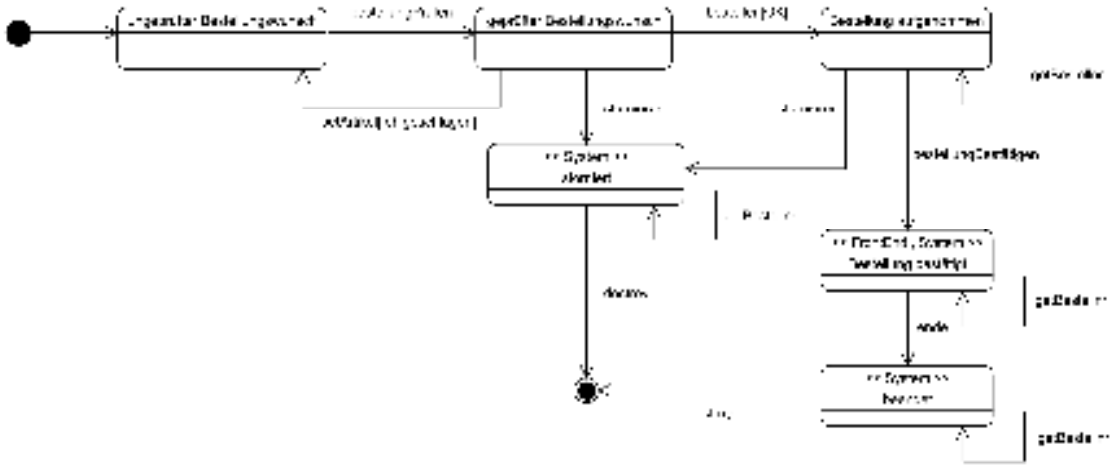

Abbildung 4-4: Referenz Zustandsdiagramm

Der nächste Schritt ist das Exportieren des Modells mit den vier Diagrammen. Dieses Modell wird, nachdem alle Werkzeuge diese Prozedur durchlaufen haben, dazu benötigt, um die Import-Funktion zu testen. Jedes Werkzeug muss jedes exportierte Modell der anderen Werkzeuge importieren. Somit entsteht eine Matrix, aus der man die Kompatibilität unter den Werkzeugen ablesen kann. Durch diese Prozedur lassen sich die Anforderungen „Interoperabilität“, „Metamodell“, „Diagrammtypen“ und ein Teil der „wichtigen (UML) Notationen“ einheitlich bewerten.

Die übrigen Anforderungen wurden durch Testen des Werkzeugs empirisch ermittelt.

4.2.4 Übersicht

Tabelle 4-4: Übersicht über alle Modellierungswerkzeuge die auch im Detail evaluiert wurden.

	Werkzeug	ArgoUML	StarUML	Fujaba	Umbrello	BOUML	Taylor MDA	Gaphor
Allgemein	Version	0.22	5.0	4.3.1	1.5.1	2.16	0.0.3	0.8.1
	Datum	15.08.2006	16.08.2006	17.08.2006	21.08.2006	21.08.2006	22.08.2006	23.08.2006
	Lizenz	BSD	GPL	LGPL	GPL	GPL	LGPL	GPL
	Bemerkungen							
Interoperabilität	Modellierungssprache	UML 1.4	UML 1.4	UML ?	UML 1.5	UML 2.0	UML 2.0	UML 2.0
	Import	XMI 1.x	XMI 1.1 UML1.3	XMI? UML1.3	XMI 1.x	Rational Rose	-	-
	Export	XMI 1.2 UML1.4	XMI 1.1 UML1.3	XMI? UML1.3	XMI 1.2 UML1.3	XMI 1.2 UML1.4	-	XMI 1.2 UML1.4
Diagrammtypen	Anwendungsfall	✔	✔	✔	✔	✔	✔	✔
	Klassen	✔	✔	✔	✔	✔	✔	✔
	Aktivität	✔	✔	✔	✔	in nächster Version	✔	✔
	Zustand	✔	✔	✔	✔	✔		
	Kompositionsstruktur		✔					
	Sequenz	✔	✔	✔	✔	✔		
	Kollaboration	✔	✔	✔	✔	✔		
	Komponenten	✔	✔		✔	✔		
	Verteilung	✔	✔		✔	✔		
Wichtige Notationen	Stereotypen	✔	✔ 1)	✔	✔ 1)	✔ 1)	✔	
	Eigenschaftswerte	✔	✔			✔		✔
	Einschränkungen	✔	✔	✔		✔		
	UML-Profile		✔					
Sonstiges	Teamunterstützung					eingeschränkt	über Eclipse	
	IDE Integration	Eclipse Plugin in entwicklung		zusätzlicher Eclipse Plugin (alpha)			Eclipse Plugin	
	Betriebssystem	Java 1.4+	Windows	Java 1.4+	Linux (KDE)	Linux/Unix/Solaris, Mac OS X, Windows	Eclipse abhängig	Unix, Mac OS X, Windows

1) nur ein Stereotyp pro Element möglich

Tabelle 4-5: Übersicht über alle Modellierungswerkzeuge die nicht im Detail evaluiert wurden.

	Werkzeug	DOME	UMLGraph	astade	Jupe	Visual UML	Violet UML	UniMod	green	MERMAID	simpleUML
Allgemein	Version	5.3	4.4	0.5.1	0.0.7	0.2.1	0.17.3	1.3.1.36	2.5RC2	2.4.8	0.1a
	Datum	21.08.2006	21.08.2006	21.08.2006	22.08.2006	22.08.2006	22.08.2006	22.08.2006	22.08.2006	23.08.2006	23.08.2006
	Lizenz	GPL	BSD	GPL	GPL	GPL	GPL	LGPL	EPL	GPL	GPL
	Bemerkungen	Unausgereift; Schlechte Usability	Deklarative Erstellung von UML Modellen, es muss Pseudocode geschrieben werden	Keine grafischen Diagramme; Klassen, Zustände und Komponenten möglich	Sehr frühe alpha Version, visuelles Modellieren noch fast nicht möglich	Installation funktioniert nicht richtig; Darstellung der Elemente fehlerbehaftet	Frühe beta Version; eher nur zum Zeichnen geeignet	Probleme mit der Kompatibilität zu neueren Eclipse Versionen	Installation war nicht erfolgreich	Proprietäres CASE-Werkzeug; Nicht intuitiv bedienbar; Keine echte visuelle Modellierung	Sehr frühe alpha Version, die nur Klassen Zeichnen kann
Interoperabilität	Modellierungssprache	UML ?	UML ?	UML ?	UML 2.0	UML 2.0	UML ?	FSM	UML ?	UML ? / FSM	UML ?
	Import	-	?	-	-	-	-	-	-	XMI ?	-
	Export	-	?	-	XMI ?	-	-	proprietäres XML	-	XMI 1.2	-
Diagrammtypen	Anwendungsfall	✔					✔				
	Klassen	✔	✔		✔	✔	✔	✔	✔	✔	✔
	Aktivität	✔									
	Zustand	✔					✔	✔		✔	
	Kompositionsstruktur										
	Sequenz	✔	✔				✔				
	Kollaboration	✔									
	Komponenten										
	Verteilung										
Wichtige Notationen	Stereotypen	✔ 1)									
	Eigenschaftswerte										
	Einschränkungen										
	UML-Profile										
Sonstiges	Teamunterstützung				über Eclipse	über Eclipse	über Eclipse	über Eclipse	über Eclipse		
	IDE Integration				Eclipse Plugin	Eclipse Plugin	Eclipse Plugin	Eclipse Plugin	Eclipse Plugin		
	Betriebssystem	Solaris, Windows	Java 1.4+	Windows	Eclipse abhängig	Eclipse abhängig	Eclipse abhängig	Eclipse abhängig	Eclipse abhängig	Unix, Windows	Windows

1) nur ein Stereotyp pro Element möglich

Tabelle 4-6: Kompatibilitätsmatrix der Modellierungswerkzeuge

		Import							
		ArgoUML	**StarUML**	**Fujaba**	**Umbrello**	**BOUML**	**Taylor**	**Gaphor**	***Referenz***
Export	**ArgoUML**	✔	1,3,4,5,6)	✘	2,4,7,8,9)	✘	✘	✘	✔
	StarUML	1,4,7)	✔	✘	2,4,7,8,9)	✘	✘	✘	1,4)
	Fujaba	✘	2,4,7)	✔	2,7,8)	✘	✘	✘	✘
	Umbrello	1,2,7)	2,4,7)	✘	✔	✘	✘	✘	1,2,7)
	BOUML	✘	✘	✘	✘	✔	✘	✘	✘
	Taylor	✘	✘	✘	✘	✘	✔	✘	✘
	Gaphor	✘	1,7,11,12)	✘	7,8,11,12)	✘	✘	✔	7,8,11,12,13)
	Referenz	✘	2,7)	✘	2,4,7,8,9)	✘	✘	✘	✔

1) Assoziationen nicht erkannt
2) Zustandsdiagramme nicht erkannt
3) Datentypen nicht erkannt
4) Stereotypen nicht erkannt
5) Aktivitätsdiagramm: Entscheidungspunkte werden als Startpunkte erkannt
6) Zustandsdiagramm: bei Aufrufen wird der Name der Funktion nicht erkannt
7) Aktivitätsdiagramme nicht erkannt
8) Multiplizitäten teilweise falsch erkannt
9) Links bei Anwendungsfällen nicht erkannt
10) Multiplizitäten nicht erkannt
11) Anwendungsfalldiagramm nicht erkannt
12) Interfaces nicht erkannt
13) Assoziationsklasse nicht richtig erkannt

4.2.5 Werkzeuge im Detail

4.2.5.1 ArgoUML

ArgoUML ist ein klassisches UML Designwerkzeug mit kognitiver Unterstützung.

Tabelle 4-7: ArgoUML Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Interoperabilität	3	5	15
Metamodell	4	4	16
Diagrammtypen	5	4	20
Wichtige (UML) Notationen	4	3	12
Teamunterstützung	0	3	0
Support	4	3	12
Usability	3	3	9
IDE Integration	1	1	1
		Gesamtergebnis:	**85**

- **Interoperabilität:** ArgoUML liegt bei der Interoperabilität im Mittelfeld der Werkzeuge. Importieren kann das Werkzeug noch nicht besonders gut. Der Mechanismus ist sehr starr und nicht robust genug, bei den kleinsten Abwei-

chungen wird der Import abgebrochen. Das exportierte XMI ist dafür besser, viele Werkzeuge kommen damit zurecht.

- **Metamodell:** UML 1.4 ist das Metamodell von ArgoUML. Es ist gut implementiert.
- **Diagrammtypen:** Vorbildlich, es werden alle UML Diagramme und deren Elemente unterstützt. Lediglich Kommentare lassen sich nicht an allen Elementen anbringen, beispielsweise an Assoziationen.
- **Wichtige (UML) Notationen:** Es werden mehrfache Stereotypen und Eigenschaftswerte für alle Elemente unterstützt. Einschränkungen können für die wichtigsten Elemente definiert werden. Hierfür bietet ArgoUML auch einen Syntaxassistenten, der aber nur von mäßigem Nutzen ist.
- **Teamunterstützung:** Es existiert keine Teamunterstützung.
- **Support:** Hinter ArgoUML steht eine aktive Community mit Forum, FAQ's, Mailingliste, Einführungstour und einer guten Dokumentation.
- **Usability:** Die Bedienung ist recht einfach und intuitiv. Etwas umständlich ist, dass Elemente nur über ein Diagramm erstellt werden können. Einzelne Stellen sind etwas umständlich zu erreichen und es gibt keine *Undo*-Funktion.
- **IDE Integration:** Ein Projekt des Google Summer of Code 2006 ist ArgoEclipse, das die Integration von ArgoUML in die Eclipse IDE verfolgt. Das Projekt steht noch am Anfang und es sind noch keine Releases verfügbar. Mehr Informationen unter http://argoeclipse.tigris.org.

Besondere Anmerkungen:

- Wie die niedrige Versionsnummer erahnen lässt, ist ArgoUML noch in einem frühen Entwicklungsstadium und bietet noch keine hundertprozentige Stabilität und Zuverlässigkeit.
- ArgoUML hat einen einfachen internen Codegenerator für einige Programmiersprachen. Das Werkzeug unterstützt dabei auch Reverse Engineering.

Fazit:

Trotz des frühen Entwicklungsstadiums ist das Werkzeug recht gut zu benutzen und kann auch im Produktivbetrieb für kleinere Projekte eingesetzt werden, vorausgesetzt man kann mit den Einschränkungen, die damit verbunden sind, leben. ArgoUML ist ein klassisches UML Modellierungswerkzeug und kann dadurch vielfältig eingesetzt werden.

4.2.5.2 StarUML

StarUML ist ein Open-Source-Projekt, um schnell flexible und erweiterbare UML / MDA Modelle zu entwickeln. Das Ziel des Projekts ist die Verdrängung von kommerziellen Werkzeugen, wie beispielsweise RationalRose oder Together.

Tabelle 4-8: StarUML Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Interoperabilität	3	5	15
Metamodell	3	4	12
Diagrammtypen	5	4	20
Wichtige (UML) Notationen	4	3	12
Teamunterstützung	0	3	0
Support	4	3	12
Usability	4	3	12
IDE Integration	0	1	0
		Gesamtergebnis:	**83**

- **Interoperabilität:** Beim Import eines nicht unterstützten XMI-Formats versucht StarUML das Modell zu konvertieren, was dem Werkzeug erstaunlich gut gelingt. Stereotypen werden dabei oft nicht erkannt. Es werden aber ab und zu zusätzliche Elemente mit merkwürdigen Namen angelegt. Das exportierte XMI wird leider nicht so gut von den anderen Werkzeugen angenommen, was sicherlich an der recht alten XMI- und UML-Version liegt.
- **Metamodell:** Das Metamodell bei StarUML ist UML1.3 und hinkt somit etwas dem Stand der Technik hinterher. Modelliert wird in UML 1.4. Laut StarUML wird UML 2 für die Modellierung benutzt, aber beispielsweise die Aktivitätselemente werden nach UML 1.4 gezeichnet.
- **Diagrammtypen:** Bei den Diagrammtypen ist StarUML vorbildlich, es werden alle UML-Diagramme unterstützt. Außer, dass keine Kommentare in den Diagrammen möglich sind, wurden keine negativen Aspekte bei StarUML gefunden.
- **Wichtige (UML) Notationen:** Es wird nur ein Stereotyp pro Element unterstützt, aber an allen Elementen sind Stereotypen möglich. Eigenschaftswerte, Einschränkungen und UML-Profile versteht StarUML ebenfalls.
- **Teamunterstützung:** Es existiert keine Teamunterstützung.
- **Support:** Die Entwicklung an StarUML ist aktiv und es existiert ein aktives Forum. Die Dokumentation und Hilfe zu dem Werkzeug ist gut und vollständig. Es werden auch Firmen gelistet, die kommerziellen Support leisten.
- **Usability:** StarUML lässt sich leicht und intuitiv bedienen. Einzelne Stellen lassen sich allerdings nur etwas umständlich erreichen.
- **IDE Integration:** Keine Integration möglich.

Besondere Anmerkungen:

- StarUML bringt einen brauchbaren Codegenerator für Forward und Reverse Engineering mit. Die wichtigsten Sprachen, wie beispielsweise Java oder C++, werden unterstützt. Hier wird aber eher ein CASE-Ansatz verfolgt, da die Erweiterung der Transformationen teilweise sehr umständlich ist.

Fazit:

Für einfachere Anwendungen ist StarUML eine gute Lösung. Der integrierte Codegenerator bietet Basisfunktionalität, ist aber für den MDA-Ansatz weniger geeignet und verfolgt eher den CASE-Ansatz. Um mit den Modellen weiterzuarbeiten, müsste die Exportfunktion aktualisiert werden, damit eine bessere Interoperabilität möglich wird.

4.2.5.3 Umbrello

Umbrello ist ein UML-Modellierungswerkzeug, dass in den grafischen Desktop KDE für Linux integriert ist.

Tabelle 4-9: Umbrello Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Interoperabilität	3	5	15
Metamodell	3	4	12
Diagrammtypen	2	4	8
Wichtige (UML) Notationen	1	3	3
Teamunterstützung	0	3	0
Support	3	3	9
Usability	3	3	9
IDE Integration	0	1	0
		Gesamtergebnis:	**56**

- **Interoperabilität:** Umbrello kommt gut mit den externen XMI-Daten zurecht. Auch das exportierte XMI ist gut von den anderen Werkzeugen angenommen worden.
- **Metamodell:** Umbrello verwendet das UML 1.3 Metamodell. Die Modellierungssprache zum Zeichnen ist allerdings UML1.5.
- **Diagrammtypen:** Es sind keine Assoziationsklassen möglich. Alle Elemente im Aktivitätsdiagramm können nur einen ausgehenden Übergang haben. Das bereitet bei Fork-Elementen natürlich Probleme, da diese sinngemäß mehr als einen ausgehenden Übergang haben müssen. Die Elemente des Aktivitäts- und Zustandsdiagramms werden nicht im Modellexplorer angezeigt, dies deutet darauf hin, dass diese Elemente nicht im UML Modell enthalten sind. Diese Vermutung wird dadurch verstärkt, dass kein anderes Werkzeug diese Model-

le importieren kann und auch in der exportierten XMI-Datei diese Elemente nicht im UML-Teil der Datei auftauchen. Sie werden in einer proprietären XMI-Erweiterung beschrieben. Somit werden die Diagrammtypen zwar unterstützt, aber nicht im UML-Modell. Aus diesem Grund werden auch Funktionen wie Guard Conditions oder Aufrufsignale nicht unterstützt.

- **Wichtige (UML) Notationen:** Außer Stereotypen, die leider bei vielen Elementen, wie beispielsweise Assoziationen, nicht möglich sind, bietet Umbrello nichts.
- **Teamunterstützung:** Es existiert keine Teamunterstützung.
- **Support:** Umbrello selbst, wie auch das ausführliche Handbuch, sind in sehr vielen Sprachen verfügbar, das ist sehr gut. Umbrello wird auch aktiv weiterentwickelt. Leider wird kein Forum angeboten.
- **Usability:** Das Werkzeug ist im Großen und Ganzen gut zu bedienen. Funktionen, wie das Umbenennen von Modellelementen im Diagramm, sind allerdings umständlich.
- **IDE Integration:** Keine Integration möglich.

Besondere Anmerkungen:

- Umbrello hat Probleme mit Unicode.
- Das Werkzeug ist nur für Linux (KDE) verfügbar.
- Umbrello beherrscht Forward- und Reverse Engineering für viele Programmiersprachen. Die Reichweite ist allerdings noch nicht sehr weit.

Fazit:

Umbrello ist aufgrund seiner unvollständigen UML-Implementierung nur sehr eingeschränkt zu empfehlen. Solange man mit Klassen- und Anwendungsfalldiagrammen zurecht kommt, ist Umbrello ausreichend. Vom MDA-Ansatz her betrachtet, ist Umbrello allerdings im Moment noch ungeeignet. Das Werkzeug hat allerdings Potenzial zu einem guten Modellierungswerkzeug heranzureifen.

4.2.5.4 Fujaba

Das Hauptziel des Fujaba Projekts ist es eine einfach zu erweiternde UML und Java Entwicklungsumgebung bereitzustellen. Das Projekt geht von der Universität Paderborn aus.

Tabelle 4-10: fujaba Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Interoperabilität	1	5	5
Metamodell	3	4	12
Diagrammtypen	2	4	8
Wichtige (UML) Notationen	1	3	3
Teamunterstützung	0	3	0
Support	2	3	6
Usability	1	3	3
IDE Integration	1	1	1
		Gesamtergebnis:	**38**

- **Interoperabilität:** Der Import von fremden Modellen funktioniert garnicht und der Export nur mangelhaft, denn es lassen sich nur Klassendiagramme exportieren. Nur wenige Werkzeuge können mit dem Export etwas anfangen.
- **Metamodell:** Das Metamodell bei Fujaba ist ebenfalls UML 1.3. Modelliert wird wahrscheinlich mit UML 1.4. Dies ist allerdings nur eine Vermutung, weil es nirgends konkrete Angaben dazu gibt.
- **Diagrammtypen:** Es ist keine Generalisierung und Realisierung möglich. In Anwendungsfalldiagrammen lassen sich keine Assoziationen erstellen.
- **Wichtige (UML) Notationen:** Stereotypen sind nur an Klassen und Assoziationen möglich, sonst bietet Fujaba leider Nichts.
- **Teamunterstützung:** Es existiert keine Teamunterstützung.
- **Support:** Es gibt eine unfangreiche Dokumentation und ein Wiki mit Anleitungen. Das Werkzeug scheint vereinzelt weiterentwickelt zu werden, jedoch nicht mehr besonders stark.
- **Usability:** Fujaba lässt sich nur sehr umständlich bedienen. Manche Funktionen, wie beispielsweise der Import/Export, sind regelrecht versteckt.
- **IDE Integration:** Es gibt ein separates Eclipse Plugin Projekt. Dieses ist allerdings noch in einem sehr frühen Entwicklungsstadium und ist daher noch nicht benutzbar, wobei die Usability einen besseren Eindruck macht, als das Hauptprodukt.

Besondere Anmerkungen:

- Fujaba bietet auch einen internen Codegenerator für Java. Aus dieser Sicht ist Fujaba aber lediglich ein CASE-Werkzeug.

Fazit:

Fujaba eignet sich nicht als Modellierungswerkzeug, da im Grunde nur das Klassendiagramm und das Zustandsdiagramm nutzbar sind und praktisch kein Import und Export möglich ist. Durch das Eclipse-Plugin könnte das Projekt einen neuen Schub bekommen, der auch recht viel versprechend aussieht.

4.2.5.5 BOUML

BOUML ist ein freies UML 2 CASE-Werkzeug mit eingebautem Codegenerator.

Tabelle 4-11: BOUML Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Interoperabilität	0	5	0
Metamodell	4	4	16
Diagrammtypen	4	4	16
Wichtige (UML) Notationen	3	3	9
Teamunterstützung	2	3	6
Support	3	3	9
Usability	3	3	9
IDE Integration	0	1	0
Gesamtergebnis:			**65**

- **Interoperabilität:** Leider kann BOUML nur Rational Rose Modelle importieren und kein XMI. Das exportierte XMI konnte kein anderes Werkzeug importieren.
- **Metamodell:** BOUML verwendet das UML 1.4 Metamodell, zur Darstellung aber UML 2.0.
- **Diagrammtypen:** Außer, dass es erst in der kommenden Version des Werkzeugs Akivitätsdiagramme geben soll, gab es keinerlei Probleme bei der Nachmodellierung der Referenzdiagramme.
- **Wichtige (UML) Notationen:** Es ist ein Stereotyp an allen Elementen möglich. Einschränkungen mittels OCL sind nur an Elementen des Zustandsdiagramms möglich. Eigenschaftswerte beherrscht BOUML ebenfalls.
- **Teamunterstützung:** Teamunterstützung bietet BOUML in eingeschränkter Weise über einen eindeutigen Identifizierer für jeden Benutzer in Form einer Umgebungsvariable. Die Verwendung eines Versionsverwaltungssystems ist ebenfalls vorgesehen, allerdings in einer sehr einfachen Form. Es ist aber zu bezweifeln, dass diese Art der Teamunterstützung im professionellen Umfeld praktikabel ist.
- **Support:** Auf der Webseite des Werkzeugs befindet sich eine ausführliche Dokumentation und eine gute Einführung in das Werkzeug. Die Entwicklung

scheint aktiv zu sein, aber es existiert leider weder Community noch ein Forum.

- **Usability:** BOUML lässt sich ziemlich einfach und intuitiv bedienen, obwohl seine Benutzeroberfläche sehr schlicht gehalten ist. Diese Schlichtheit schränkt BOUML an manchen Stellen aber auch etwas ein und einzelne Funktionen hätten besser platziert sein können, aber insgesamt macht BOUML einen guten Eindruck.
- **IDE Integration:** Keine Integration möglich.

Besondere Anmerkungen:

- Laut Hersteller ist BOUML extrem resourcensparend und sehr performant.
- Der eingebaute Generator kann C++, Java und IDL Code erzeugen und ist in eingeschränkter Form auch anpassbar. Reverse Engineering wird ebenfalls angeboten.

Fazit:

Hinter der schlichten und einfachen Oberfläche verbirgt sich ein gut nutzbares Modellierungswerkzeug. Leider fehlt es an der Interoperabilität, somit kann das Werkzeug im MDA-Kontext nicht sinnvoll eingesetzt werden.

4.2.5.6 Taylor MDA

Taylor MDA ist ein auf Eclipse basierendes MDA-Werkzeug, welches auf die Erstellung von Enterprise-Anwendungen zugeschnitten ist. Es steuert andere Open-Source-Werkzeuge wie Maven 2, JBoss Seam, und JBoss Portal.

Tabelle 4-12: Taylor MDA Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Interoperabilität	1	5	5
Metamodell	5	4	20
Diagrammtypen	2	4	8
Wichtige (UML) Notationen	2	3	6
Teamunterstützung	1	3	3
Support	3	3	9
Usability	2	3	6
IDE Integration	4	1	4
Gesamtergebnis:			**61**

- **Interoperabilität:** Taylor MDA besitzt weder Import- noch Exportfunktionen. Da das Modell allerdings im EMF-UML2-Format ist, könnte eventuell mit anderen

Eclipse Werkzeugen importiert bzw. exportiert werden. Diese Möglichkeit wurde allerdings nicht untersucht.

- **Metamodell:** Taylor MDA verwendet das Eclipse eigene UML2, das eine EMF basierte Implementation des UML 2 Metamodells ist. EMF besteht wiederum aus dem Ecore Metamodell, das auf dem Essential MOF Standard basiert. Somit erfüllt die Eclipse-eigene UML2 Lösung auch die MDA-Anforderungen der OMG.
- **Diagrammtypen:** Es ist keine Realisierung möglich und Klassen können nur Attribute und keine Operationen haben. Bei Interfaces verhält es sich genau umgekehrt. Taylor MDA hat noch viele weitere Einschränkungen, weil das Werkzeug nur die wirklich benötigten Funktionen, Elemente und Diagramme bereitstellt, die für die spezielle Anwendungszielgruppe benötigt werden.
- **Wichtige (UML) Notationen:** Jedem Element können mehrere Stereotypen zugewiesen werden. Einschränkungen und Eigenschaftswerte werden nicht angeboten.
- **Teamunterstützung:** Es können die Funktionen von Eclipse zur Teamunterstützung genutzt werden. Taylor MDA selber bietet keine speziellen Funktionen, um im Team an einem Modell zu arbeiten.
- **Support:** Auf der Webseite wird ein Einführungsbeispiel und eine Installationsanleitung angeboten um sich mit dem Werkzeug vertraut zu machen. Ein Forum wird ebenfalls angeboten und das Werkzeug wird aktiv weiterentwickelt. Leider gibt es keine Dokumentation zu dem Werkzeug.
- **Usability:** Die meisten Einstellungen lassen sich nicht direkt im Diagramm machen, man muss dazu umständlicherweise in der „Properties“-Tabelle oder in der Baumansicht navigieren.
- **IDE Integration:** Taylor MDA ist ein Eclipse Plugin und ist somit nahezu perfekt in diese IDE integriert.

Besondere Anmerkungen:

- Taylor MDA ist ein komplettes Generator-Framework mit hochgradig anpassbaren Templates. Dieser Teil wird im Abschnitt 4.3.5.5 weiter untersucht.
- Das frühe Entwicklungsstadium merkt man dem Werkzeug kaum an. Es funktioniert dafür sehr stabil und zuverlässig.

Fazit:

Da Taylor MDA speziell auf (Java) Enterprise-Anwendungen abzielt und deshalb nur die dafür notwendigen Funktionen und Elemente der UML implementiert, können die Modelle kaum für andere Generatoren verwendet werden.

4.2.5.7 Gaphor

Gaphor ist eine leicht zu benutzende und erweiterbare UML Modellierungsumgebung, die ihren Fokus auf dem Zeichnen von Diagrammen hat.

Tabelle 4-13: Gaphor Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Interoperabilität	1	5	5
Metamodell	4	4	16
Diagrammtypen	2	4	8
Wichtige (UML) Notationen	2	3	6
Teamunterstützung	0	3	0
Support	1	3	3
Usability	2	3	6
IDE Integration	0	1	0
		Gesamtergebnis:	**44**

- **Interoperabilität:** Es existiert keine Importierfunktion für XMI. Das exportierte XMI ist unsauber, es fehlt der initiale XML-Tag und der erzeugte Zeitstempel hat ein ungültiges Format. Der Zeitstempel muss manuell angepasst werden, damit ein valides XMI-Dokument entsteht, welches dann auch von anderen Werkzeugen gelesen werden könnte. Mit dem exportierten XMI haben andere Werkzeuge große Probleme. Entweder kann das XMI gar nicht oder nur das Klassendiagramm gelesen werden.
- **Metamodell:** Gaphor nutzt das UML 2 Metamodell. Schade ist allerdings, dass beim Export nur das UML1.4-Metamodell herangezogen wird.
- **Diagrammtypen:** Beim Aktivitätsdiagramm sind keine Wächterbedingungen bei Aktivitätsübergängen möglich, lediglich eine Beschreibung. Das Zustandsdiagramm wird nicht unterstützt. Klassen- und Anwendungsfalldiagramm funktionieren gut. Es gibt keine speziellen Diagrammtypen (Klassen-, Anwendungsfall-, Aktivitäts-, …), alle Elemente der verschiedenen Diagramme werden in ein Standarddiagramm gezeichnet. Deshalb können Elemente eigentlich verschiedener Diagramme in einem Diagramm zusammen erscheinen.
- **Wichtige (UML) Notationen:** Stereotypen werden theoretisch unterstützt, allerdings konnte die Funktion praktisch nicht getestet werden, weil in der Dokumentation nicht beschrieben ist, wie man Stereotypen anzuwenden hat und die Funktionsweise auch nicht intuitiv zu erkennen ist. Eigenschaftswerte werden dafür unterstützt, leider aber keine Einschränkungen.
- **Teamunterstützung:** Es existiert keine Teamunterstützung.
- **Support:** Es wird eine Onlinedokumentation mit mäßiger Qualität angeboten. Außerdem kann Gaphor nicht viel bieten. Die Entwicklung an dem Werkzeug scheint aber noch aktiv zu sein.

- **Usability:** Das Zeichnen der Diagramme ist zum Teil etwas umständlich, weil die Verbindungen zwischen Elementen, beispielsweise Assoziationen, nur an den Außenlinien der Elemente „andocken“ können. Im Übrigen lässt sich das Werkzeug einfach und intuitiv bedienen.
- **IDE Integration:** Keine Integration möglich.

Fazit:

Gopher ist für die Weiterverarbeitung der Modelle nicht besonders gut geeignet. Gopher eignet sich eher als Zeichenwerkzeug für UML 2.

4.2.6 Fazit Modellierungswerkzeuge

Aus der Evaluierung der Modellierungswerkzeuge ergibt sich ein deutliches Bild. Die Werkzeuge auf dem Open-Source-Markt sind noch nicht ausgereift und für größere Projekte eher ungeeignet. Dies resultiert hauptsächlich aus der noch fast überall fehlenden Teamunterstützung und der eher seltenen Möglichkeit zur Integration in eine Entwicklungsumgebung. Grund hierfür ist einfach das Alpha- oder Beta-Stadium der meisten Werkzeuge.

Für kleinere Projekte hingegen können Werkzeuge wie ArgoUML oder StarUML eine Alternative zu kommerziellen Werkzeugen sein. Allerdings muss dann im Einzelfall genau geprüft werden, ob das entsprechende Open-Source-Werkzeug den Anforderungen im Projekt gerecht wird.

Die Open-Source-Werkzeuge müssen noch einige Zeit reifen, dann aber könnten sie ernsthafte Optionen zu den kommerziellen Werkzeugen werden.

4.3 Evaluierung von Generierungswerkzeugen

Unter den Generierungswerkzeugen versteht man Werkzeuge, die aus Modellen Plattformspezifischen Code generieren können. Oft sind diese eng an Plattformen, wie J2EE und .NET, gekoppelt, lassen sich aber in der Regel beliebig erweitern.

4.3.1 Einschränkungen

Auch bei den Generierungswerkzeugen werden grundsätzlich nur Open Source Produkte betrachtet.

Ebenso werden nur Werkzeuge untersucht, die sich zumindest grob an den Konzepten der MDA orientieren, beispielsweise dem Konzept der plattformunabhängigen Modelle.

4.3.2 Anforderungen

Die Anforderungen an ein Generierungswerkzeug im Bezug auf die Nutzung im MDA-Kontext sind wie folgt:

- **Modelltransformation:** Wie flexibel und erweiterbar ist der Transformationsprozess? Kann das Werkzeug auch Modell-zu-Modell Transformationen durchführen und unterstützt es Reverse Engineering? Welche Plattformen unterstützt der Generator von Haus aus und wie ist die Codequalität? Wie und mit welchen Mitteln lassen sich Transformationen modifizieren oder neu erstellen?
- **Interoperabilität:** Das Werkzeug sollte möglichst auf gängigen Standards aufbauen. Optimal wären die, welche die OMG in der MDA-Spezifikation vorschlägt. Beispielsweise wäre es gut, wenn die Transformationsregeln unabhängig vom Werkzeug wären und sich somit mit jedem Werkzeug verwenden ließen oder wenn sie Modelle im XMI Format verarbeiten könnten.
- **Erweiterbarkeit:** Das Werkzeug muss sich flexibel und möglichst einfach erweitern lassen. Hier sind UML-Profile oder Metamodellierung eine Möglichkeit. Ebenso ist die Unterstützung von Einschränkungs- oder Aktionssprachen ein wichtiger Punkt, um im Modell unabhängig von der Plattform bestimmte nicht modellierbare Dinge zu spezifizieren.
- **Artefaktgenerierung:** Das Werkzeug sollte manuell geschriebenen Code in einer gewissen Weise schützen können, damit der Generator diesen nicht überschreibt. Ebenso sollte ein Debug-Mechanismus zur Verfügung stehen, um Fehlerquellen schnell identifizieren zu können. Das Werkzeug sollte auch bei der Erstellung der Dokumentation und von Testfällen unterstützen.
- **Teamunterstützung:** Im Gegensatz zu den Modellierungswerkzeugen, muss bei den Generierungswerkzeugen gemeinsam am generierten Code weitergearbeitet werden können und auch die Generator-Konfiguration verwaltet werden können. Das Werkzeug muss hier nicht unbedingt selbst die Teamunterstützung bereitstellen. Dies kann beispielsweise auch durch die Integration in eine IDE von dieser übernommen werden.
- **Support:** Es ergeben sich ähnliche Anforderungen an den Support wie bei den Modellierungswerkzeugen. Hinzu kommen hier aber noch Einführungsbeispiele, da die Komplexität der Generatoren deutlich höher ist als die der Modellierungswerkzeuge und durch entsprechende Beispiele ein schnellerer Einstieg möglich ist.

- **Modellierung:** Das Generierungswerkzeug beeinflusst maßgeblich die Modellierung. In wie weit kann das Werkzeug abstrahieren und wie orientiert es sich dabei an den MDA-Richtlinien? Unter dieser Anforderung fallen auch die Unterstützung einer Modellverifikation bevor das Modell transformiert werden kann und die Unterstützung für Markierungen im Modell (Stereotypen oder Eigenschaftswerte).
- **Usability:** Das Werkzeug sollte sich intuitiv und möglichst einfach bedienen lassen. Dies reduziert die Einarbeitungsphase, spart Zeit bei der täglichen Bedienung und erhöht die Akzeptanz bei den Anwendern. Zur Usability gehören aber auch die Stabilität, Skalierbarkeit und Performance des Werkzeugs.
- **Integration:** Das Werkzeug sollte sich in den bestehenden Entwicklungsprozess integrieren lassen. Dazu gehört die Integration in die bestehende Entwicklungsumgebung, beispielsweise Eclipse oder Microsofts Visual Studio, aber auch in ein Buildwerkzeug wie Ant oder Maven.

4.3.2.1 Gewichtung der Anforderungen

- **Modelltransformation (5):** Die Modelltransformation ist das Kerngeschäft eines Generators, daher ist diese Anforderung unverzichtbar und wird mit fünf gewichtet.
- **Interoperabilität (4):** Die Interoperabilität bei den Generierungswerkzeugen ist nicht so fundamental wichtig, wie bei den Modellierungswerkzeugen, da der Generator üblicherweise das letzte Glied in der MDA-Pipeline ist. Interoperabilität beschreibt auch einen gewissen Grad von Werkzeugunabhängigkeit. Solche Abhängigkeiten haben sich langfristig als sehr problematisch herausgestellt. Daher die Gewichtung mit vier.
- **Erweiterbarkeit (4):** Da die Erweiterbarkeit auch die Reichweite und Flexibilität des Generierungswerkzeugs bestimmt, wird diese mit vier gewichtet.
- **Artefaktgenerierung (4):** Da die Artefaktgenerierung ein unumgänglicher Teil im Projektalltag ist, aber den MDA-Prozess nur teilweise betrifft, wird diese Anforderung mit vier gewichtet.
- **Teamunterstützung (4):** Die Unterstützung eines Mehrbenutzerbetriebs ist für fast jedes Projekt unverzichtbar. Daher wird die Teamunterstützung mit vier gewichtet.
- **Support (4):** Der Support spielt bei den Generatoren eine noch wichtigere Rolle als bei den Modellierungswerkzeugen, weil der Einstieg in ein Generierungswerkzeug deutlich schwieriger und somit eine gute Unterstützung sehr wichtig ist. Aus diesem Grund die Gewichtung mit vier.
- **Modellierung (3)**: Da die Modellierungsanforderungen nur teilweise den MDA-Prozess beeinflusst wird diese Anforderung mit drei gewichtet.

- **Usability (3):** Durch eine gute Usability wird das Werkzeug schnell akzeptiert und es macht Spaß, damit zu arbeiten. Die Usability hat wenig Einfluss auf den MDA-Ansatz, deshalb die Gewichtung mit drei.
- **Integration (2):** Die Integration in die bestehende Umgebung ist wichtig für das reibungslose Funktionieren des Entwicklungsprozesses. Ebenso wird der tägliche Umgang mit dem Werkzeug erleichtert, daher die Gewichtung mit zwei.

4.3.3 Vorgehen

In diesem Abschnitt möchte ich mein Vorgehen während der Evaluierung der Generierungswerkzeuge beschreiben.

Am Anfang bin ich dabei ähnlich Vorgegangen, wie zuvor bei den Modellierungswerkzeugen. Als Erstes habe ich die folgenden Quellen nach Werkzeugen durchsucht und ebenfalls in einer Tabelle aufgelistet.

Quellen:

- http://www.modelbased.net/mda_tools.html
- http://sourceforge.net
- http://freshmeat.net
- http://google.de

Der nächste Schritt war das Anreichern der Tabelle mit Informationen zu den Werkzeugen von deren Webseiten. Dabei bin ich auf weitere Werkzeuge gestoßen.

4.3.4 Übersicht

Tabelle 4-14: Übersicht über alle evaluierten Generierungswerkzeuge

	Werkzeug	oAW	AndroMDA	Acceleo	openMDX	Taylor MDA	JAG	pmMDA
Allgemein	Version	4.1	3.2-Snapshot	1.1.0	1.12.1	0.0.3	6.1	0.3 RC3
	Datum	14.09.2006	08.09.2006	29.09.2006	18.09.2006	20.09.2006	25.09.2006	27.09.2006
	Lizens	EPL	BSD	GPL	BSD	LGPL	GPL	Apache v2.0
Interoperabilität	Modell	UML1.4, EMF-UML2, EMF, …	UML 1.4 (EMF-UML2)	UML 1.x, UML2, EMF-UML2, …	UML 1.4	EMF-UML2	UML 1.4	UML 1.4
	Metamodell	Ecore,MOF, XML DOM, …	MOF, Ecore	Ecore, MOF, UML(2), …	MOF	UML2	UML	UML
	Import	XMI, EMF, Visio, Text, XML, …	XMI, EMF, …	XMI 1.1, 1.2, 2.0, EMF, EMF-UML2	XMI, Rose, Together, …	EMF-UML2	XMI 1.2	ArgoUML (XMI)
	Export	✗	✗	EMF-UML2	✗	EMF-UML2	XMI 1.2	✗
Erweiterbarkeit	UML-Profile	✓	✓	✓	✗ 1)	✓	✓	✓
	Metamodellierung	✓	✓	✓	✗ 1)	✗	✗	✗
	Aktions- / Einschränkungs-sprachen	Check (OCL ähnlich)	OCL	✗	✗	✗	✗	✗
Modellierung	Abstraktionsebene	aPIM, PIM, PSM (beliebig)	aPIM	PIM, aPIM	PIM	aPIM	aPIM	aPIM
	Unterstützung für Markie-rungen	✓	✓	✓	✗ 1)	✓	✓	✓
	Modellverifikation	✓	✓ 2)	✓ 2)	✗	✗	✓ 2)	✓ 2)
Transformation	Transformationssprache	Xpand(2)	Velocity (austauschbar)	Eigene Template Sprache	Java	JET	Velocity	Velocity
	Forward Engineering	-	J2EE, RCP, .NET, Struts, JSF, XML, Hibernate , EJB, Spring, …	(Struts, EJB, Hibernate, Web Services, Eclipse RCP, .Net. C#, PHP, Doku) 5)	J2SE, J2EE, .NET, EJB, CORBA	J2EE, EJB3,	J2EE, EJB, Hibernate, Spring, Tapestry, Xfire, Axis, Struts, …	J2EE, CORBA, .NET,
	Reverse Engineering	✗	DB-Schema	✗	✗	DB-Schema	DB-Schema	✗
	Modell-zu-Modell	über Xtend (austauschbar)	in v.4.0 (voraussichtlich ende 2006)	✗	✗	✗	✗	✗
	Modifikation/Erstellung von Transformationen	✓	✓	✓	✓	✓	✓	✓
Artefaktgenerie-rung	Dokumentationsgenerierung	✗	✗	✗	✗	✗	✗	✗
	Debug-Unterstützung	✓	✓	✓	✓	✓	✗	✗
	Testfallgenerierung	✗	✗	✗	✗	JUnit	JUnit	✗
	Codeschutz-Bereiche	✓	● 3)	✓	✗ 1)	● 4)	✗	✗
Support	Comunity	aktiv	aktiv	aktiv	aktiv	aktiv	aktiv	✗
	Forum	gut	sehr gut	gut	mittelmäßig	mittelmäßig	gut	mittelmäßig
	Mailinglist/Newsgroup	✓	✓	✗	✓	✓	✗	✓
	Dokumentation	sehr gut	sehr gut	gut	gut	mittelmäßig	mittelmäßig	schlecht
	Tutorials/Beispiele	sehr gut	sehr gut	gut	Tutorial nicht aktuell, Beispiele funktionieren nicht wie beschrieben	gut	gut	mittelmäßig
Sonstiges	Teamunterstützung	über Eclipse	über Eclipse oder Visual Studio	über Eclipse	über Eclipse	über Eclipse	über IDE	über IDE
	IDE Integration	Eclipse Plugin	Eclipse Plugin, Visual Studio,	Eclipse Plugin	Projekte können in Eclipse importiert werden	Eclipse Plugin	Projekte können in IDE importiert werden	Projekte können in IDE importiert werden
	Build Integration	Ant, Maven2 Plugin (For-nax)	Ant, Maven(2) Plugin	✗	Ant	Maven2	✗	✗
	Betriebssystem	Eclipse abhängig	Java 1.4+	Eclipse abhängig	Java 1.4+	Eclipse abhängig	Java 1.4+ (GUI für Windows + Linux/Unix)	Java 1.5+

1) in Konzept nicht benötigt

2) vor der Transformation wird validiert

3) auf Datei-Ebene, durch ableiten der generierten Klasse

4) Java Dateien werden gemerged, statische Non-Java Dateien bleiben unangetastet (z.B. pom.xml), von andere Non-Java Dateien wie JSPs werden Sicherungskopien erstellt.

5) verfügbar in Acceleo Pro (kommerziell)

4.3.5 Werkzeuge im Detail

4.3.5.1 openArchitectureWare (oAW)

openArchitectureWare ist ein sehr flexibles, praxisorientiertes und modular aufgebautes MDA/MDSD Generator Framework, das komplett in die Eclipse IDE integriert ist und ein Bestandteil des Eclipse GMT (Generative Modeling Tools) Projekts ist.

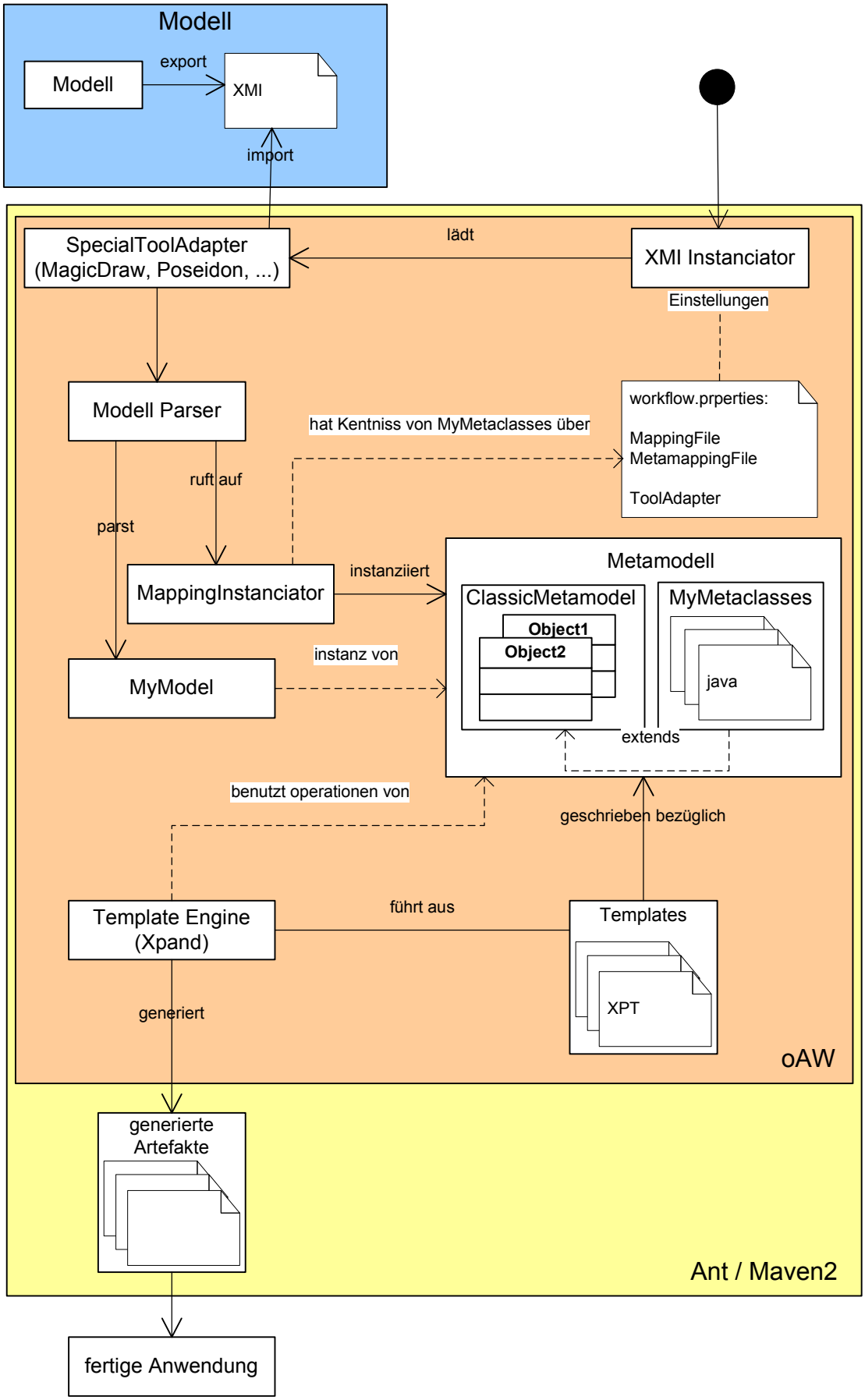

Abbildung 4-5: oAW Architektur

Tabelle 4-15: oAW Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Modelltransformation	4	5	20
Interoperabilität	5	4	20
Erweiterbarkeit	4	4	16
Artefaktgenerierung	3	4	12
Teamunterstützung	3	4	12
Support	4	4	16
Modellierung	4	3	12
Usability	4	3	12
Integration	4	2	8
Gesamtergebnis:			**128**

- **Modelltransformation:** Im Lieferumfang von openArchitectureWare sind keine Cartridges enthalten. oAW vertritt die Philosophie, nur für das eigentliche Generator-Framework Support zu leisten und sieht die Cartridges als Erweiterungen, die jeder selbst implementieren muss. Es ist allerdings geplant, zentral über die oAW Website auf Drittanbieter von Cartridges zu verweisen. Das Werkzeug unterstützt sowohl Modell-zu-Code-Transformationen mittels der Templatesprache Xpand als auch Modell-zu-Modell Transformationen über die funktionale Sprache Xtend. Für diese Sprachen wird ein Texteditor geboten, der Syntax highlighting, intelligente Codevervollständigung und Fehler-Markierungen unterstützt. Das Besondere an den Sprachen die in oAW verwendet werden ist, dass sie auf dem gleichen Typensystem basieren und dieselbe Auszeichnungssprache verwenden und außerdem statisch typisiert sind.
- **Interoperabilität:** oAW ist ein extrem flexibles Werkzeug. Es kann mit fast jeder Art von Modellierungswerkzeug umgehen, dazu muss nur ein entsprechender ToolAdapter für ein bestimmtes Werkzeug implementiert werden. Adaptoren für gängige Werkzeuge, wie Magic Draw und Poseidon oder auch für EMF und Eclipse UML2, werden mitgeliefert. Das Werkzeug kann allerdings mit den Modellen ohne das dazu passende Metamodell nicht arbeiten. Hierzu wird das verwendete Metamodell oder UML-Profil über die Workflowkonfiguration bekannt gemacht. openArchitectureWare kann mit jedem beliebigen Metamodell arbeiten. Somit können beispielsweise auch textuelle, domänenspezifische Sprachen (DSL) als Metamodell herangezogen werden. Von Haus aus kommt oAW mit Ecore und MOF basierten Metamodellen klar.
- **Erweiterbarkeit:** openArchitectureWare ist die Erweiterung schlechthin. Durch die extreme Flexibilität, zum einen durch die Unterstützung der UML basierten Modellierung inklusive UML-Profile, zum anderen durch die Alternative ein eigenes Metamodell zu verwenden. Auch eine OCL-ähnliche Sprache namens Check wird für die deklarative Beschreibung von Einschränkungen angeboten. Darüber hinaus ist über diese Sprache auch eine Modellvalidierung möglich. Eine Aktionssprache wird aber nicht angeboten.

- **Artefaktgenerierung:** Zum Schutz von manuell geschriebenem Code werden in oAW so genannte *Protected Regions* eingesetzt. Diese Bereiche werden über das Template definiert und im Workflow kann dann konfiguriert werden, welche Verzeichnisse und Dateitypen berücksichtigt werden sollen. Um den Workflow zu debuggen, kann der Debug-Mechanismus der Eclipse IDE benutzt werden. Für die Erstellung von Tests oder Dokumentation gibt es keine Unterstützung, aufgrund der Philosophie von oAW. Natürlich kann das aber über eine Cartridge umgesetzt werden.

- **Teamunterstützung:** openArchitectureWare bietet selbst keine Unterstützung für den Mehrbenutzerbetrieb, aber durch die enge Kopplung an Eclipse liegt es nahe, die Funktionalität dieser Entwicklungsumgebung zu nutzen.

- **Support:** oAW bietet einen weitreichenden und guten Support. Es wird eine umfangreiche und gut strukturierte Dokumentation, eine Newsgroup, eine FAQ-Sektion und viele Beispiele und Anleitungen von der aktiven Community bereitgestellt. Es wird auch auf Firmen verwiesen, die kommerziell Support und Beratung anbieten. Leider passt das etwas klein geratene Forum nicht ganz in das bisher sehr gute Support-Bild.

- **Modellierung:** Es können einerseits Markierungen oder Eigenschaftswerte für die Transformation von UML-Modellen verwendet werden, beispielsweise annotierte PIMs, andererseits kann auch bei der Verwendung eines eigenen Metamodells die Semantik dessen für die Transformation verwendet werden. Der Abstraktionsgrad kann beliebig gewählt werden. Es kann durch die Modell-zu-Modell Transformation von einem CIM über ein PIM und PSM bis zum Code generiert werden. Allerdings muss alles selber definiert werden. oAW bietet dazu keine Funktionalität, nur seine Generierungsfähigkeiten. Modellverifikation ist nur bedingt möglich. oAW verwendet dazu das recipe-Framework. Damit ist es beispielsweise möglich, während des Ant-Builds die generierten Artefakte auf Vollständigkeit zu prüfen oder zu prüfen, ob eine Subklasse einer bestimmten Basisklasse existiert. Diese Art der Verifikation ist besonders für manuell zu erweiternde Teile geeignet.

- **Usability:** Durch die sehr gute Integration in die Eclipse IDE lässt sich das Werkzeug sehr gut von einer Stelle aus bedienen. Wird dazu noch ein EMF-basiertes Modell verwendet, muss man Eclipse für die komplette Entwicklung vom Modell bis zur fertigen Anwendung nicht verlassen. Zur Stabilität des Werkzeugs lässt sich sagen, dass es absolut stabil und zuverlässig arbeitet, auch in großen Projekten wurde es schon eingesetzt. Große Modelle verarbeitet oAW ohne Probleme, nur die Performance bricht dabei langsam ein. Es gibt Ansätze für eine parallele Verarbeitung der Workflows, dies steigert die Performance beispielsweise auf Doppel(kern)prozessor-Systemen um bis zu 40%. In zukünftigen Versionen könnte dies implementiert sein.

- **Integration:** Das Werkzeug ist leider nur als Eclipse Plugin verfügbar, was natürlich auch eine Abhängigkeit zu der IDE mit sich bringt. Dafür wird man aber mit einer beispielhaften Integration belohnt. oAW ist nahezu perfekt in die Entwicklungsumgebung Eclipse integriert und nutzt die gegebene Infrastruktur sehr gut aus. Das Werkzeug kann auch über ein Maven2-Plugin oder über Ant gesteuert werden, somit werden die wichtigsten Buildwerkzeuge unterstützt.

Fazit:

openArchitectureWare ist ein extrem flexibles und erweiterbares Generatorframework. Es eignet sich besonders für große Projekte, die mit DSLs oder eigenen Metamodellen arbeiten wollen, aber auch für Projekte, die einen langsamen Einstieg in die modellgetriebene Entwicklung wollen. Kleinere Projekte sind mit dem Werkzeug eher nicht empfehlenswert, da keine Cartridges mitgeliefert werden und deshalb der Aufwand beim ersten Projekt, in Relation zur Projektgröße, überaus groß ist.

4.3.5.2 AndroMDA

AndroMDA ist ein sehr vielfältiges Open-Source-MDA-Generator-Framework, das für einfache CRUD Anwendungen bis hin zu komplexen Enterprise-Anwendungen eingesetzt werden kann.

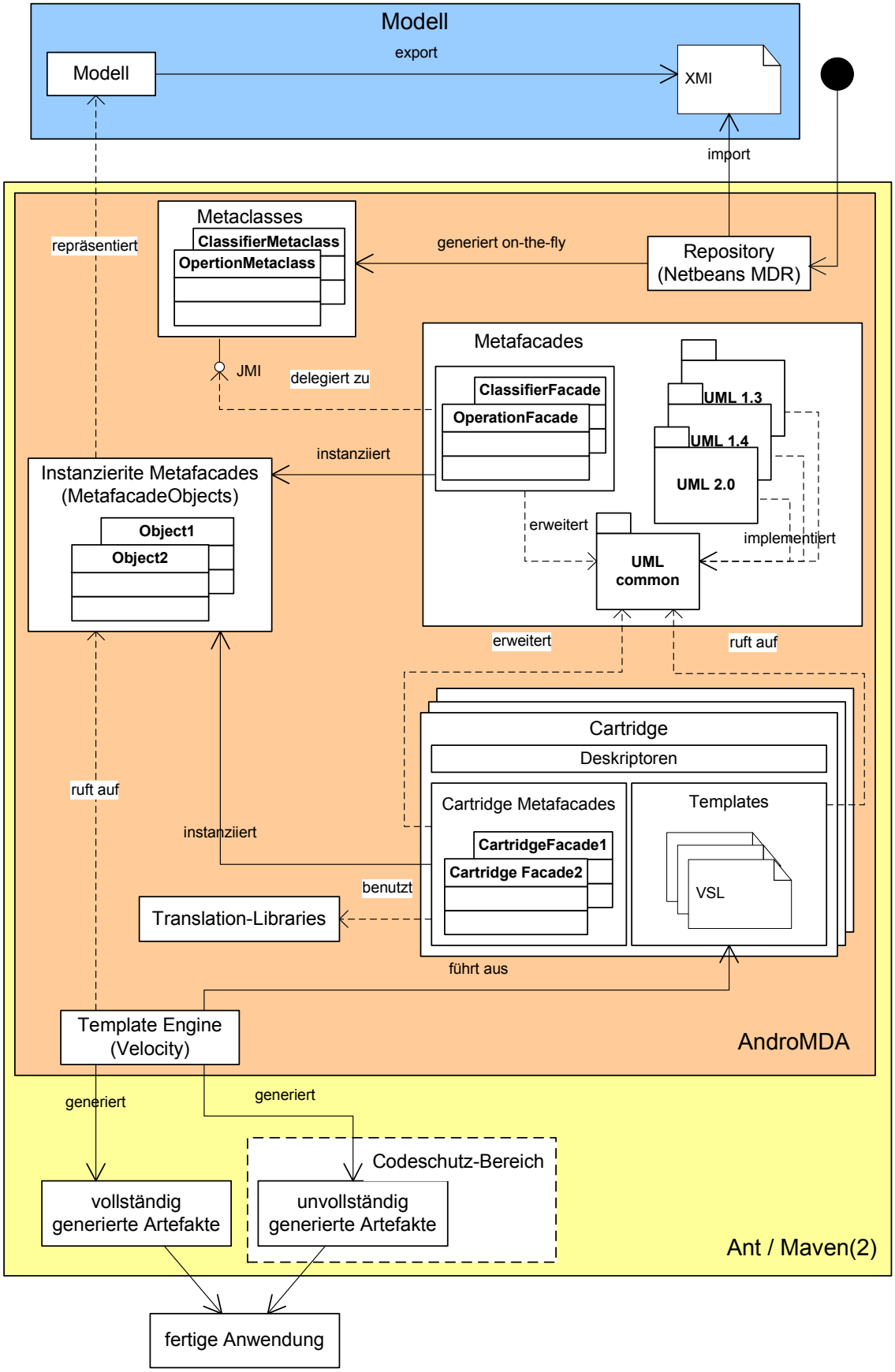

Abbildung 4-6: AndroMDA Architektur

Tabelle 4-16: AndroMDA Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Modelltransformation	4	5	20
Interoperabilität	4	4	16
Erweiterbarkeit	4	4	16
Artefaktgenerierung	3	4	12
Teamunterstützung	3	4	12
Support	5	4	20
Modellierung	3	3	9
Usability	3	3	9
Integration	3	2	6
Gesamtergebnis:			**120**

- **Modelltransformation:** Das Werkzeug bringt von Haus aus schon viele qualitativ hochwertige Cartridges mit, welche für die Transformation vom annotierten PIM in plattformspezifischen Code verantwortlich sind. Diese Cartridges können aber auch erweitert oder selbst erstellt werden. Eine Cartridge ist bei AndroMDA ein JAR-Archiv und besteht aus Templates, Metafacades (Java-Klassen) und mehreren Deskriptoren im XML Format. Metafacade steht für „Metamodel Facade“. Diese Metafacades stellen eine Schnittstelle, auf vom Repository erzeugte Modellelemente bereit und kapseln so die konkrete Implementierung des Metamodells ab. Templates können nur mit den Metafacades arbeiten, dadurch werden sie sehr viel einfacher, weil die ganze Verantwortung für die Transformation und Generierung bei zentralen Java-Objekten liegt und nicht bei der Template-Sprache. Die Entwicklung von Metafacades wird durch die AndroMDA-Meta-Cartridge unterstützt. Dabei wird ein Modell, welches das UML-Metamodell und die bereits im AndroMDA-Kern implementierten Metafacades umfasst, als Ausgangspunkt für Erweiterungen genutzt. Über die Templates wird das so zusagen im Speicher existierende PSM in Code abgebildet. Dafür stellt AndroMDA die Template Engine Velocity bereit. Diese kann aber auch ausgetauscht werden, genau wie das Repository. AndroMDA bietet in der voraussichtlich 2007 erscheinenden Version 4 unter anderem auch Modell-zu-Modell Transformationen. Reverse Engineering ist nur eingeschränkt über ein zusätzliches Werkzeug für Datenbank-Schemata möglich.
- **Interoperabilität:** AndroMDA verarbeitet UML Modelle im XMI-Format oder EMF-basierte Modelle. Das Repository, welches die entsprechenden Modelle lädt und instanziiert kann aber auch ersetzt oder erweitert werden, um auch Modelle anderer Metamodelle, die nicht MOF oder Ecore basierend sind, zu verarbeiten. AndroMDA hält sich also ziemlich gut an die Vorgaben der OMG.
- **Erweiterbarkeit:** Erweitern lässt sich AndroMDA durch UML-Profile und auch mit etwas höherem Aufwand über Metamodellierung. Ein weiterer Erweiterungsmechanismus sind die Translation-Libraries. Sie können von den Cartridges

genutzt werden, um im Modell in OCL formulierte Ausdrücke in Code zu übersetzten. Eine Aktionssprache wird nicht unterstützt.

- **Artefaktgenerierung:** AndroMDA geht einen relativ ungewöhnlichen Weg um manuell hinzugefügten Code zu schützen. Dabei wird von einer generierten Basisklasse abgeleitet und diese wird dann in einem anderen Pfad abgelegt. Diese abgeleitete Klasse wird dann vom Generator nicht wieder überschrieben. Diese Methode ist allerdings fehleranfällig, wenn beispielsweise die Basisklasse gelöscht oder umbenannt wird, weil sich das Modell geändert hat oder man Artefakte generieren will, die keine Generalisierung kennen. In der Praxis hat sich diese Methode allerdings bewährt. Eine spezielle Unterstützung für den Test oder die Dokumentation gibt es nicht. Es kann aber eine Cartridge erweitert oder neu erstellt werden, um diese Funktionen bereitzustellen. Debug-Unterstützung gibt es nur über das Buildwerkzeug. Es werden aussagekräftige Fehlermeldungen ausgegeben.

- **Teamunterstützung:** Eine explizite Unterstützung wird von AndroMDA nicht angeboten, allerdings kann die Teamunterstützung der Entwicklungsumgebung genutzt werden, wenn AndroMDA in eine solche integriert ist.

- **Support:** Der Support von AndroMDA ist beispielhaft. Es wird ein großes, aktives Forum geboten, eine Mailingliste, eine kleine FAQ Sektion und eine gute, weitreichende Dokumentation für das Produkt selber, als auch für die zahlreichen Transformationen (Cartridges) und Erweiterungen. Natürlich werden auch einige Beispiele und Anleitungen bereitgestellt. Sogar kommerzielle Beratung und Trainings durch den Gründer des Projekts werden angeboten.

- **Modellierung:** AndroMDA benutzt ein annotiertes PIM als Input und erkennt über Stereotypen und Eigenschaftswerte, welche Cartridges wie getriggert werden müssen. Das PSM entsteht dabei nur virtuell im Speicher und wird somit übersprungen und es wird direkt zu Code transformiert. Vor der Transformation wird allerdings eine Modellvalidierung durchgeführt, dabei wird das Modell gegen das Metamodell und alle Anforderungen der beteiligten Cartridges geprüft. Die Anforderungen die AndroMDA an ein Modellierungswerkzeug stellt, sind normal und können von den meisten Werkzeugen am Markt erfüllt werden.

- **Usability:** Die Bedienung von AndroMDA ist an vielen Stellen nicht eben komfortabel. Da das Werkzeug auf anderen Werkzeugen wie Ant oder Maven aufbaut, ist die Einrichtung der Umgebung etwas langwierig, aber nicht besonders schwierig. Die Erstellung eines neuen Projektes unterstützt ein Projektkonfigurationsskript. Damit werden alle benötigten Cartridges identifiziert und die komplette Projektumgebung erstellt. Die Bedienung funktioniert größtenteils über das Buildwerkzeug, eine grafische Bedienoberfläche ist aber in Entwicklung (siehe Integration). Ist alles eingerichtet, wird AndroMDA über das Buildwerkzeug gesteuert und die Bedienung gelingt somit gut. AndroMDA ist

schon in größeren Projekten eingesetzt worden. Dabei hat sich gezeigt, dass es sehr zuverlässig funktioniert und für den produktiven Einsatz geeignet ist. Wenn allerdings große Modelle mit vielen Cartridges verwendet werden, kann der Transformationsprozess sehr lange dauern.

- **Integration:** Es existiert ein Plugin für die Entwicklungsumgebung Visual Studio. Für Eclipse ist ein Plugin, das aber schon über eine Update-Seite installiert werden kann, noch in Entwicklung. Die Plugins sollen eine komplette grafische Oberfläche für AndroMDA darstellen und beispielsweise Funktionen für die Projekterstellung oder das Editieren von Templates bieten. AndroMDA wird fast komplett über ein Ant- oder Maven-Plugin gesteuert und ist somit sehr gut in den Buildprozess integriert. Bei der Erzeugung der Projektumgebung werden auch automatisch alle benötigten Dateien für Eclipse erzeugt. Somit kann das Projekt einfach in Eclipse importiert werden.

Fazit:

Dank der vielen Qualitativ hochwertigen Cartridges kommt man mit AndroMDA sehr schnell zu einem brauchbaren Ergebnis, dazu trägt auch der sehr gute Support bei. Die Software ist ausgereift und kann problemlos im produktiven Einsatz verwendet werden. Auch für große Projekte ist AndroMDA geeignet, dank der flexiblen Erweiterungsmechanismen, die das Framework bereitstellt.

4.3.5.3 Acceleo

Acceleo ist ein Codegenerator, der dazu entwickelt wurde, effizient Software mittels MDA zu entwickeln und die Produktivität bei der Entwicklung zu steigern.

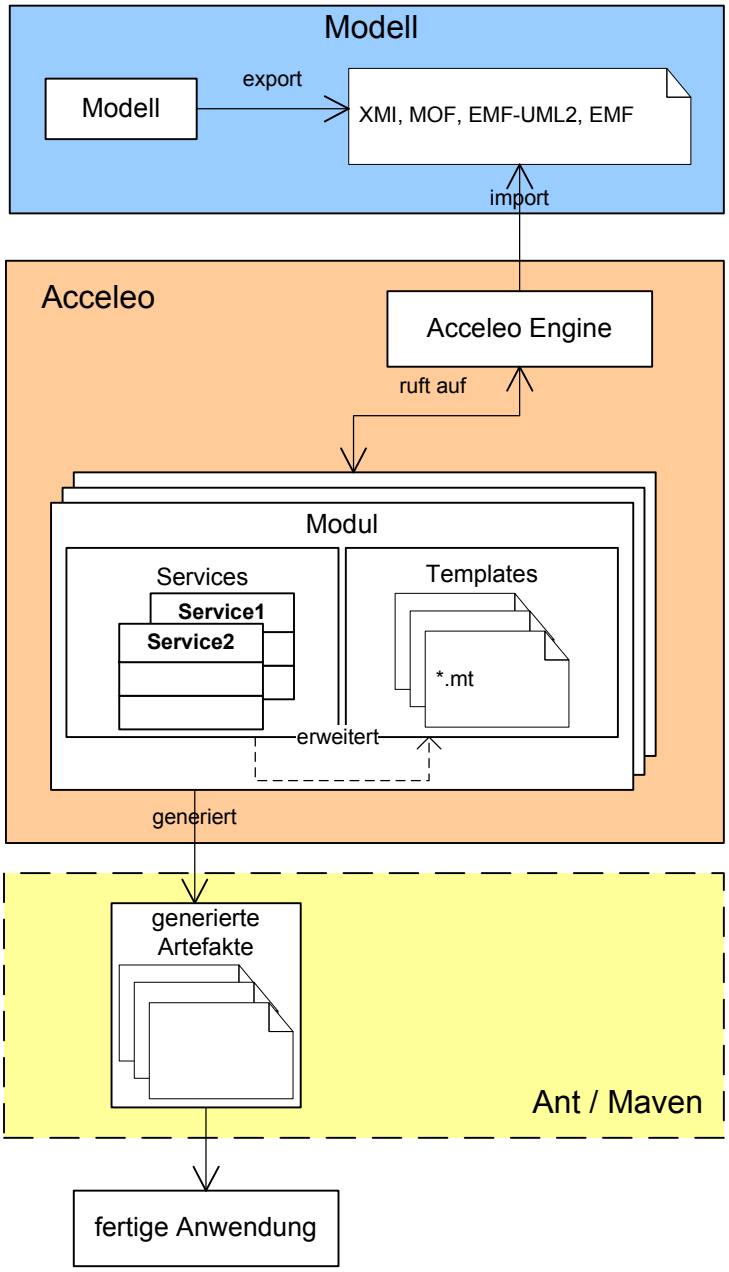

Abbildung 4-7: Acceleo Architektur

Tabelle 4-17: Acceleo Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Modelltransformation	4	5	20
Interoperabilität	4	4	16
Erweiterbarkeit	3	4	12
Artefaktgenerierung	3	4	12
Teamunterstützung	3	4	12
Support	4	4	16
Modellierung	4	3	12
Usability	4	3	12
Integration	4	2	8
		Gesamtergebnis:	**120**

- **Modelltransformation:** Acceleo verpackt seine Transformationsbeschreibungen in Module, welche als Eclipse Generator Projekt umgesetzt werden. Die Module beinhalten Templates, die in einer eigenen proprietären Sprache geschrieben werden und so genannte Services, die mittels Java implementiert werden und Funktionen für die Templates bereitstellen. Von Haus aus bringt Acceleo keine Module mit. In einem Beispiel sind allerdings einige Module für DAO, DTO oder ein Webfrontend auf JSP / Servlet Basis enthalten. Für das Erstellen der Templates ist ein Editor mit Syntax Highlighting, Fehlererkennung und Autovervollständigung integriert. Modell-zu-Modell-Transformationen oder Reverse-Engineering unterstützt das Werkzeug nicht.
- **Interoperabilität:** Das Werkzeug kommt mit fast allen Arten von Modellen und Metamodellen zurecht. Die Transformationsbeschreibungen mit der proprietären Sprache ist aus Sicht der Interoperabilität ein Nachteil. Die Sprache ist aber sehr einfach und leicht zu erlernen und stellt deshalb keine größere Hürde dar.
- **Erweiterbarkeit:** Mittels Metamodellierung oder UML-Profilen lässt sich Acceleo einfach erweitern. Eine Aktions- oder Einschränkungssprache unterstützt es aber leider nicht. Solche Sprachen sind auch im Konzept des Werkzeugs nicht vorgesehen, da hier die Philosophie so aussieht, dass zwar so viel wie möglich modelliert werden soll, aber nur bis zu einem gewissen Punkt, da ab dort eine konventionelle Codierung praktikabler ist.
- **Artefaktgenerierung:** Manuell hinzugefügter Code kann mit Acceleo geschützt werden, wenn in den Templates bereits ein Bereich eingefügt wird, der manuell geschrieben Code enthalten soll. Somit kann dann im generierten Artefakt in diesen Bereichen beliebiger Code manuell hinzugefügt werden. Fehler beim Erstellen der Templates werden durch den Editor schon hervorgehoben. Zusätzlich kann eine Vorschau eines Templates in Echtzeit angezeigt werden, wenn ein entsprechendes Modellelement markiert wird. Somit ist die Entwicklung und das Debuggen von Templates sehr einfach und schnell. Für die Generierung der Dokumentation wird ein kommerzielles Modul angeboten, für den Test existiert so etwas leider nicht.
- **Teamunterstützung:** Das Werkzeug bietet selbst keine Teamunterstützung, kann aber leicht durch die perfekte Eclipse-Integration von der IDE übernommen werden. Dies wird auf der Website von Acceleo auch empfohlen.
- **Support:** Es wird ein gutes Forum, eine gute Dokumentation und ein gutes Einführungsbeispiel angeboten. Beim Support wird allerdings eine besondere Strategie verfolgt. Sollte der Bereitgestellte Support nicht ausreichen, können bei der Entwicklerfirma aus Frankreich zusätzlich Dokumentation, Support, Beratung oder Module kommerziell erworben werden. Ebenso wird noch eine kostenpflichtige *Pro*-Version des Werkzeugs angeboten, die unter anderem

folgende Zusatzfunktionen mitbringt: Modell-Code-Modell Synchronisation, automatischer Code merge und Reporting für Projektleiter.

- **Modellierung:** Acceleo unterstützt sehr viele Modellierungswerkzeuge. Die Modellierung wird quasi nicht eingeschränkt, Acceleo kann sich sehr gut anpassen. Modelle können Markierungen in Form von Stereotypen aufweisen, müssen aber nicht. Somit können PIMs mit und ohne Markierungen verarbeitet werden und in beliebige Artefakte transformiert werden. Eine Verifikation der Modelle ist leider nicht möglich, es wird nur vor der Transformation eine Validierung des Modells durchgeführt.
- **Usability:** Die Bedienung des Werkzeugs ist anfangs etwas gewöhnungsbedürftig, aber ist dann sehr einfach. Besonders das Entwickeln von Templates geht sehr leicht von der Hand, da der Editor eine sehr gute Unterstützung bietet. Dank der sehr guten Integration in Eclipse passt alles hervorragend zusammen und wirkt wie aus einer Form gegossen. Das Werkzeug hat sich schon in großen und kleinen Projekten bewährt und läuft absolut zuverlässig und performant.
- **Integration:** Acceleo ist nahezu perfekt in die Eclipse IDE integriert. Leider kann der Transformationsprozess nicht aus einem Buildwerkzeug angestoßen werden, jedoch die generierten Artefakte können natürlich problemlos über ein Buildwerkzeug verwaltet werden.

Fazit:

Acceleo eignet sich besonders, wenn man kommerziellen Support haben will.

4.3.5.4 openMDX

openMDX ist ein offenes Entwicklungs- und Integrationsframework, welches eine hoch anpassbare Plugin-Architektur bereitstellt. Das Werkzeug ist eine generische, modellgetriebene Laufzeitplattform für verteilte, komponentenbasierte und serviceorientierte Anwendungen und setzt die MDA auf eine sehr spezielle Weise um.

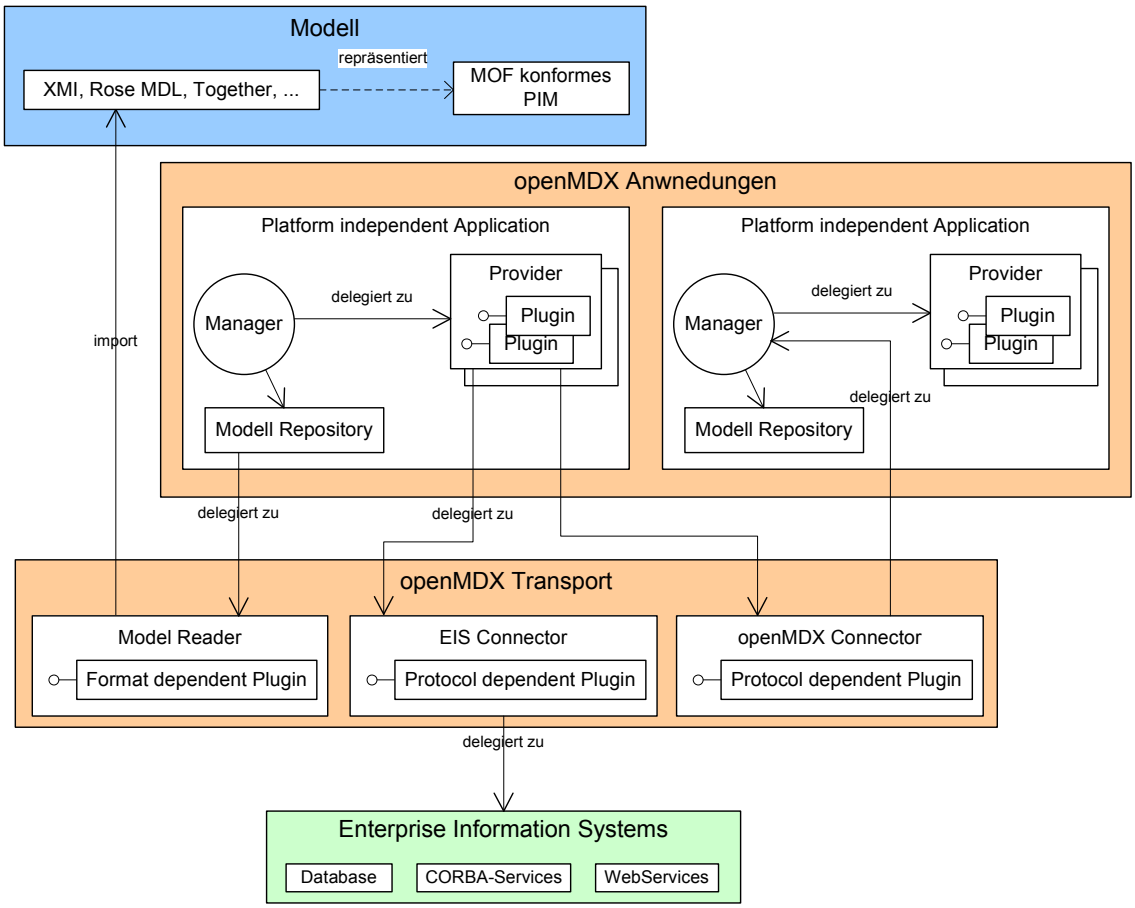

Abbildung 4-8: openMDX Architektur

OpenMDX ist eine Laufzeitumgebung, die sich zwischen der zu entwickelnden Applikation und den darunter liegenden Plattformen platziert. Die Anwendungslogik wird in Form von Plugins für die Laufzeitumgebung in Java entwickelt und ist somit zielplattformunabhängig.

Tabelle 4-18: openMDX Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Modelltransformation	3	5	15
Interoperabilität	4	4	16
Erweiterbarkeit	2	4	8
Artefaktgenerierung	3	4	12
Teamunterstützung	2	4	8
Support	3	4	12
Modellierung	4	3	12
Usability	3	3	9
Integration	3	2	6
		Gesamtergebnis:	**98**

- **Modelltransformation:** Aus dem Modell werden erweiterte JMI Interfaces und Zugriffsklassen, mit Hilfe der in JMI definierten MOF-JMI-Mappings generiert. Die Anwendungslogik wird in Form von Plugins für die openMDX Plattform manuell implementiert und nutzt dazu die generierten JMI Klassen und Inter-

faces. Diese Plugins sind plattformunabhängige Plugins, da diese auf der abstrakten OpenMDX Laufzeitumgebung basieren. Die Laufzeitumgebung regelt dann über eine Konfiguration die Zielplattform. Über plattformspezifische Plugins kann die Laufzeitumgebung um neue Zielplattformen erweitert werden. Modell-zu-Modell Transformationen werden in diesem Konzept nicht benötigt und sind deshalb auch nicht implementiert. Reverse Engineering ist mit diesem Konzept auch nicht möglich, da kein Code für eine spezielle Plattform generiert wird, sondern erst zur Deployment-Zeit der Anwendung die Zielplattform konfiguriert wird. Durch dieses Konzept ist es allerdings nicht möglich, für eine beliebige Programmiersprache Code zu erzeugen. Man ist immer an Java gebunden und muss mit dessen Facetten leben. Beispielsweise sind Echtzeitanwendungen kaum umsetzbar.

- **Interoperabilität:** An die von der OMG vorgeschlagenen Standards hält sich openMDX sehr genau. Modelle sollten in UML modelliert werden, müssen aber MOF konform sein. Importiert wird im Normalfall XMI, aber auch andere Formate können mittels Plugins gelesen werden. Der Zugriff auf die Modelldaten geschieht über den JMI-Standard. Die Transformationen werden allerdings in normalem Programmcode in Java geschrieben. Somit ist man stark an diese Sprache und ihre Facetten gebunden.
- **Erweiterbarkeit:** openMDX ist an allen wichtigen Stellen durch entsprechende Plugins erweiterbar, UML-Profile oder Metamodellierung passen aber nicht ins Konzept von openMDX und werden daher nicht unterstützt. Eine Einschränkungs- oder Aktionssprache gibt es leider auch nicht.
- **Artefaktgenerierung:** Codeschutzbereiche sind im Konzept nicht nötig, da die generierten JMI Repräsentation um die Businesslogik erweitert werden muss. Dieser Mechanismus basiert ebenfalls auf Vererbung, ähnlich wie bei AndroMDA. Ein Debuggen ist nur über Eclipse oder Ant möglich, da es sich hier um ein Framework handelt und keine Standalone-Anwendung. Es können an einigen Stellen auch Debug-Parameter gesetzt werden. Dokumentation oder Testfälle deckt das Framework nicht ab, es kann aber beispielsweise manuell über Javadoc Dokumentation generiert, oder mittels JUnit Testfälle definiert werden.
- **Teamunterstützung:** Teamunterstützung bietet das Werkzeug selber nicht, kann aber beispielsweise von Eclipse übernommen werden.
- **Support:** OpenMDX nutzt für seinen Support (Forum, Mailingliste) die durch Sourceforge.net bereitgestellte Infrastruktur. Ebenfalls wird ein sehr kleiner FAQ-Bereich bereitgestellt. Kommerziellen Support und Workshops werden durch die OMEX AG, die das Framework entwickelt hat und andere Firmen bereitgestellt. Die Beschreibung zum Einführungsbeispiel ist über ein Jahr alt und sollte dringen aktualisiert werden, da sie nicht mehr synchron zu dem bereitgestellten Code / Produkt ist.

- **Modellierung:** Das Werkzeug stellt keine besonders hohen Anforderungen an ein UML-Modellierungswerkzeug. Da im Konzept keine Markierungen, wie Stereotypen oder Eigenschaftswerte nötig sind, erfüllen die meisten Modellierungswerkzeuge die Anforderungen. Es muss nur das reine PIM Modelliert werden und im XMI-Format bereitgestellt werden. Es werden Werkzeuge wie IBM Rational Software Modeler, Rational Rose, Magic Draw, Poseidon oder Together unterstützt. Eine Modellverifikation ist leider nicht möglich.
- **Usability:** Die Bedienung, beziehungsweise das ganze Konzept, ist nicht völlig transparent. Es erfordert eine relativ lange Einarbeitungszeit, um das Werkzeug zu verstehen, da sich das Konzept stark von den anderen Werkzeugen unterscheidet. OpenMDX hat sich auch schon in größeren Projekten bewährt und ist somit für den produktiven Einsatz geeignet. Das recht große Projekt openCRX, welches ein CRM-System implementiert, wird als Beispiel mit allen Modellen und Source Code bereitgestellt. Bei der Stabilität und der Performance gibt es keine wirklichen Einschränkungen, außer dass, bedingt durch die Laufzeitumgebung, ein gewisser Performanceverlust für die zu entwickelnde Anwendung entsteht. Erwähnenswert ist noch, dass durch das spezielle Konzept sehr kurze Roundtrip-Zeiten erreicht werden, da keine klassischen Modelltransformationen nötig sind.
- **Integration:** OpenMDX wird normalerweise über Ant oder die Entwicklungsumgebung Eclipse gesteuert. Eine grafische Oberfläche wird allerdings nicht angeboten.

Besondere Anmerkungen:

- Das Framework ist stark auf Java-Enterprise-Anwendungen fixiert.
- Es ist kein Wechsel zur klassischen Entwicklung möglich, wodurch das Risiko für das Projekt deutlich erhöht wird.

Fazit:

openMDX ist das einzige Werkzeug, dass keinen generativen Ansatz verfolgt, sondern auf einer plattformunabhängigen Laufzeitumgebung aufbaut. Auf den ersten Blick scheint der Ansatz einige Vorteile zu haben, beispielsweise die sehr späte Festlegung auf die Zielplattform, aber bei genauerem Betrachten stellt sich heraus, dass zuerst eine lange Einarbeitungsphase in die Umgebung nötig ist, bevor das Werkzeug sinnvoll genutzt werden kann.

4.3.5.5 Taylor MDA

Taylor MDA ist ein auf Eclipse basierendes MDA Werkzeug, welches auf die Erstellung von Enterprise-Anwendungen zugeschnitten ist.

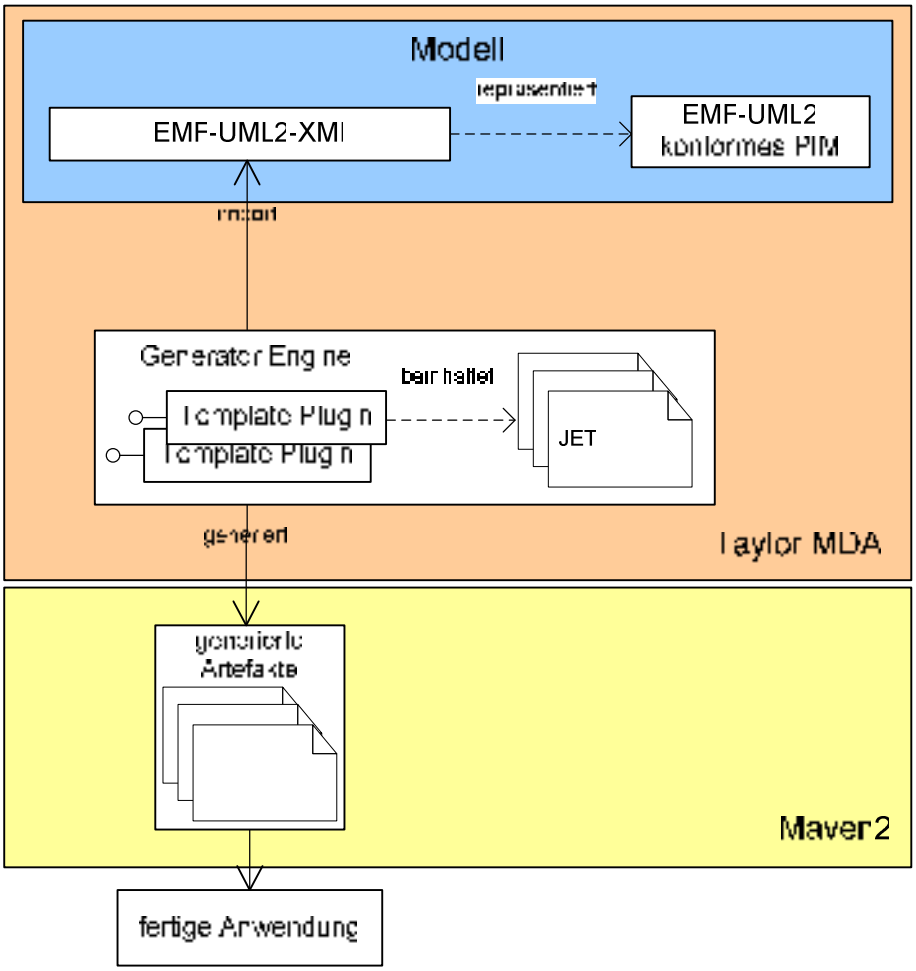

Abbildung 4-9: Taylor MDA Architektur

Tabelle 4-19: Taylor MDA Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Modelltransformation	4	5	20
Interoperabilität	2	4	8
Erweiterbarkeit	2	4	8
Artefaktgenerierung	3	4	12
Teamunterstützung	3	4	12
Support	2	4	8
Modellierung	4	3	12
Usability	2	3	6
Integration	3	2	6
		Gesamtergebnis:	**92**

- **Modelltransformation:** Transformationen werden mittels JET-Templates spezifiziert, welche sich modifizieren und erweitern lassen. Neue Templates werden mittels Eclipse-Plugins hinzugefügt. Über Eclipse kann der Pfad zu den

Templates verändert werden. Modell-zu-Modell Transformationen unterstützt das Werkzeug nicht. Taylor MDA kann aus Datenbank-Schemen ein UML-Modell erstellen. Der Zugriff auf die Datenbank erfolgt mittels JDBC. Von Haus aus generiert das Werkzeug unter anderem EJB3 Code und die dazugehörigen Deployment Deskriptoren, die Maven2 Projektumgebung, JUnit Tests und noch Einiges mehr. Die Qualität der generierten Artefakte ist brauchbar und man kann gut darauf aufbauen.

- **Interoperabilität:** Taylor MDA kann keine fremden Modelle einlesen, sondern nur EMF-UML2 Modelle verarbeiten. Es ist so ausgelegt, dass kein anderes externes Modellierungswerkzeug verwendet werden muss. Ansonsten verwendet das Werkzeug das UML2-Metamodell.
- **Erweiterbarkeit:** Mittels UML-Profile lässt sich das Werkzeug erweitern. Metamodellierung wird allerdings nicht unterstützt. Einschränkungs- oder Aktionssprachen bietet Taylor MDA leider auch nicht.
- **Artefaktgenerierung:** Einen Schutz des manuell hinzugefügten Codes löst Taylor MDA auf verschiedene Weise, abhängig vom Artefakt. Java Code wird gemerged, statische Dateien wie die pom.xml oder application.xml werden übersprungen und andere nicht-Java Dateien wie JSPs, werden mittels Backup gesichert. Eine Debug-Unterstützung wird über Eclipse oder Maven angeboten. Unterstützung bei der Dokumentation gibt es voraussichtlich erst in einem zukünftigen Release. Für die Unterstützung beim Test generiert Taylor MDA automatisch JUnit Tests für Session Beans.
- **Teamunterstützung:** Teamunterstützung gibt es über die Eclipse IDE. Dank der guten Integration in Eclipse liegt dies aber auch sehr nahe.
- **Support:** Für den Support werden die von Sourceforge.net bereitgestellten Ressourcen (Forum und Mailingliste) verwendet. Die Entwicklung ist aktiv, aber es existiert noch keine richtige Dokumentation. Ein gutes Einführungsbeispiel ist vorhanden, aber weiterführende Konzepte, wie beispielsweise das Erweitern der Templates, werden nicht beschrieben.
- **Modellierung:** Da das Modellierungswerkzeug in Taylor MDA mit inbegriffen ist, gibt es keine Probleme bei der Übergabe der Modelle an den Generator. Das Modellierungswerkzeug beherrscht exakt die Funktionen, die auch wirklich benötigt werden, somit ist die Bedienung einfach. Es können auch andere Modellierungswerkzeuge verwendet werden, wenn diese EMF-UML2 Modelle erzeugen können. Der Generierungsprozess wird ausschließlich über Stereotypen gesteuert. Somit erhält der Generator ein annotiertes PIM. Die Stereotypen werden dann während des Generierungsprozesses in Java-Annotationen umgesetzt. Eine explizite Modellverifikation ist nicht möglich.
- **Usability:** Die Benutzung ist dank des integrierten Modellierungswerkzeugs einfach. Der Generierungsprozess ist vergleichsweise langsam. Für den Produk-

tivbetrieb ist das Werkzeug noch nicht ausgereift genug. Erfahrungen mit größeren Projekten existieren deshalb noch nicht.

- **Integration:** Taylor MDA ist sehr gut in Eclipse integriert. Die benötigten Daten für das Buildwerkzeug Maven2 werden zwar generiert, aber der Generierungsprozess kann nicht von Maven2 angestoßen werden.

Fazit:

Taylor MDA ist das einzige Werkzeug, dass ein Modellierungswerkzeug beinhaltet. Somit ist dieses Werkzeug eine gute „All-in-one“ Lösung für Java-Enterprise-Anwendungen. Leider befindet es sich noch in einer frühen Entwicklungsphase und ist somit für den Produktivbetrieb eher ungeeignet.

4.3.5.6 JAG

JAG (Java Application Generator) ist eine reine Java Applikation, generiert unter anderem die komplette Projektumgebung und ist speziell für J2EE-Anwendungen optimiert.

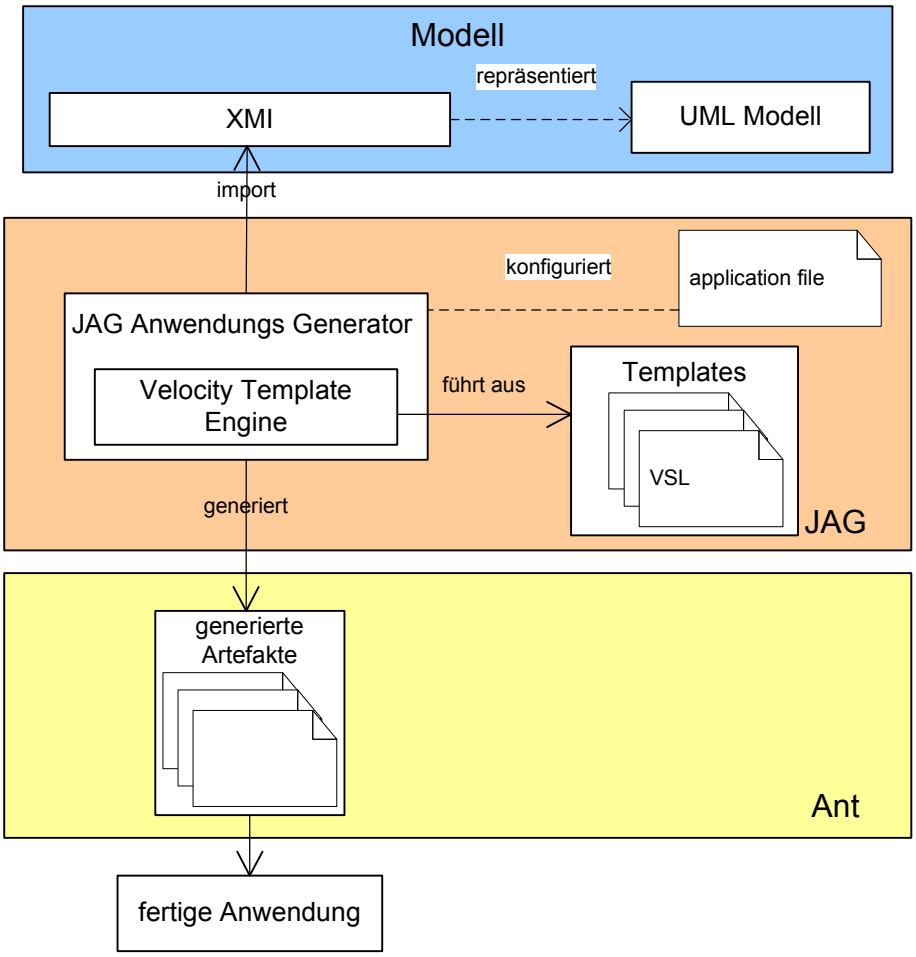

Abbildung 4-10: JAG Architektur

Tabelle 4-20: JAG Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Modelltransformation	4	5	20
Interoperabilität	3	4	12
Erweiterbarkeit	2	4	8
Artefaktgenerierung	2	4	8
Teamunterstützung	2	4	8
Support	2	4	8
Modellierung	3	3	9
Usability	4	3	12
Integration	1	2	2
		Gesamtergebnis:	**87**

- **Modelltransformation:** Das Werkzeug bringt eine Vielzahl an fertigen Templates für beispielsweise EJB, Hibernate, Spring, Struts oder Ant mit, welche leicht angepasst und erweitert werden können. Die Qualität der generierten Artefakte ist sehr hochwertig. Die mitgelieferten Templates entstanden aus langjähriger praktischer Erfahrung und basieren auf vielen Best Practices aus dem J2EE-Bereich. Geschrieben werden die Templates in der Velocity Template Language (VTL), welche dann von einer integrierten Velocity Engine zur Generierungszeit ausgeführt werden. JAG kann damit beliebige Artefakte generieren, ist aber sehr auf Java-Enterprise-Anwendungen fixiert. Über eine Konfigurationsdatei (application file) können die zu verwendenden Templates ausgewählt werden, somit steuert man den Generator. Reverse Engineering wird für Datenbank Schemata unterstützt, Modell-zu-Modell Transformationen allerdings nicht.
- **Interoperabilität:** JAG liest UML Modelle im XMI 1.2 Format ein und kann diese auch wieder in diesem Format exportieren. Das ist besonders praktisch, wenn ein bestehendes Datenbank-Schema eingelesen wird und mittels UML-Modellierung erweitert werden soll. Als Metamodell wird das UML 1 Metamodell herangezogen.
- **Erweiterbarkeit:** Über UML-Profile kann JAG erweitert werden, Metamodellierung wird aber nicht angeboten. Einschränkungs- oder Aktionssprachen werden nicht unterstützt.
- **Artefaktgenerierung:** Manuell hinzugefügter Code kann nicht geschützt werden. Beim Generierungsprozess wird allerdings geprüft, ob das zu generierende Artefakt schon existiert. Ist das der Fall, so wird ein Dialog angezeigt, wie weiter verfahren werden soll. Zur Wahl steht „Überschreiben“, „nicht Überschreiben“ und „Unterschiede anzeigen“. Wenn die Artefakte überschrieben werden sollen, erstellt JAG automatisch Sicherheitskopien der betroffenen Dateien, die dann manuell oder mit Hilfe eines externen Werkzeugs gemerged werden können. Einen Debug-Modus hat das Werkzeug nicht. JUnit Testfälle werden dafür automatisch generiert. Bei der Dokumentation bietet JAG keine Unter-

stützung, dies kann aber in Form eines entsprechenden Templates hinzugefügt werden.

- **Teamunterstützung:** Eine Teamunterstützung gibt es nicht. Diese Funktion muss beispielsweise über die verwendete IDE, wie Eclipse oder NetBeans, abgedeckt werden.
- **Support:** Das Forum basiert auf dem Sourceforge.net Forum und ist vergleichsweise aktuell. Eine Mailingliste oder Newsgroup gibt es nicht. Eine Beispielanwendung, an der die Funktionsweise erklärt wird, ist enthalten und durchaus brauchbar. Leider besteht die Dokumentation fast ausschließlich aus der Beschreibung der Beispielanwendung.
- **Modellierung:** JAG verlangt ein annotiertes PIM und generiert daraus direkt Code. Die Markierungen werden mittels Stereotypen gemacht, Eigenschaftswerte werden nicht verwendet. Fast jedes Modellierungswerkzeug kann diese Anforderungen erfüllen. Es wird vor dem Generierungs-Prozess eine Validierung des Modells vorgenommen. Dabei werden einige vordefinierte Regeln abgeprüft.
- **Usability:** Die Benutzung von JAG ist sehr einfach. Dank der grafischen Oberfläche kann man nicht viel falsch machen. Es gibt auch nicht sehr viele Optionen, man hat sich auf das Wesentliche beschränkt. Es wird jedoch kein Editor zur Bearbeitung der Templates mitgeliefert. Das Werkzeug hat sich schon in größeren Projekten bewährt. Bei der Performance könnte es eventuell Probleme geben, da auch schon sehr kleine Modelle relativ lange für die Generierung der Artefakte benötigen. Da aber nach jedem Roundtrip nicht das komplete Projekt neu generiert wird, fällt das nicht so sehr ins Gewicht.
- **Integration:** JAG lässt sich weder in eine Entwicklungsumgebung, noch in ein Buildwerkzeug integrieren. Entwicklungsumgebungen können das generierte Projekt zwar recht einfach integrieren, aber es wird keine Unterstützung auf Seiten von JAG dafür bereitgestellt. JAG kann zwar alle Artefakte für das Buildwerkzeug Ant generieren, aber diese können nicht den Transformations-Prozess anstoßen.

Besondere Anmerkungen:

- Das Werkzeug ist sehr stark auf J2EE und Java fixiert, daher ist eine Umstellung auf andere Plattform wie .Net nicht empfehlenswert.

Fazit:

JAG ist ein Werkzeug, speziell für Java-Enterprise-Anwendungen. Für andere Plattformen ist das Werkzeug eher ungeeignet. Wegen der mangelnden Dokumentation ist das Werkzeug nur bedingt empfehlenswert.

4.3.5.7 pmMDA

pmMDA (poor man MDA) ist ein Werkzeug, das eine pragmatische Umsetzung der MDA für kommerzielle und industrielle Projekte darstellt.

Tabelle 4-21: pmMDA Evaluierungsergebnisse

Anforderung	Bewertung	Gewichtung	
Modelltransformation	3	5	15
Interoperabilität	1	4	4
Erweiterbarkeit	2	4	8
Artefaktgenerierung	1	4	4
Teamunterstützung	2	4	8
Support	1	4	4
Modellierung	3	3	9
Usability	4	3	12
Integration	1	2	2
		Gesamtergebnis:	**66**

- **Modelltransformation:** pmMDA verwendet, wie einige andere Werkzeuge, die Velocity-Template-Sprache, um die Transformationen zu beschreiben. Die Erweiterung der Templates ist somit relativ einfach, ein Editor wird dafür aber nicht mitgeliefert. Von Haus aus bringt das Werkzeug einige Transformationstemplates mit, unter anderem für J2EE, CORBA, .NET und EJB. Die Qualität jener ist recht gut. Der Transformationsprozess läuft im Prinzip wie bei JAG ab. Eine Modell-zu-Modell Transformation oder Reverse Engineering ist nicht möglich.
- **Interoperabilität:** Das Werkzeug ist verhältnismäßig starr, was den Import von Modellen angeht. Es kommt nur mit Modellen älterer ArgoUML- oder Poseidon-Versionen zurecht. Generell unterstützt es aber UML-Modelle im XMI Format.
- **Erweiterbarkeit:** Mittels UML-Profilen lässt sich pmMDA erweitern. Metamodellierung, Aktions- oder Einschränkungssprachen werden allerdings nicht unterstützt.
- **Artefaktgenerierung:** Nachträglich hinzugefügter Code lässt sich nicht schützen. Der Generator überschreibt Änderungen ohne vorheriges nachfragen. Einen Debug-Modus gibt es nicht. Fehlermeldungen werden in der Oberfläche angezeigt, allerdings ohne detaillierte Informationen. Unterstützung für die Erstellung der Dokumentation oder beim Test bietet pmMDA nicht. Es können allerdings Templates für diese Zwecke erstellt werden.
- **Teamunterstützung:** Das Werkzeug selbst hat keine Unterstützung für einen Mehrbenutzerbetrieb. Diese Funktion kann aber beispielsweise durch die verwendete IDE bereitgestellt werden.

- **Support:** Beim Support schneidet pmMDA schlecht ab. Es gibt noch keine Community, das Forum und die Mailingliste bieten kaum Informationen, die Dokumentation ist sehr kurz und unvollständig und die Einführungsbeispiele funktionieren nicht oder sind kaum dokumentiert.
- **Modellierung:** pmMDA verwendet, wie die meisten heutigen Generierungswerkzeuge, Stereotypen und Eigenschaftswerte, um den Generierungsprozess zu steuern. Als Input wird also ein annotiertes PIM erwartet, woraus dann direkt Code generiert wird. Diese Anforderungen können viele Modellierungswerkzeuge umsetzen. Vor einer Transformation führt pmMDA noch eine Validierung des Modells durch. Dieser Prozess kann nicht beeinflusst werden.
- **Usability:** Da pmMDA nicht mit vielen Funktionen protzt, ist auch die Bedienung des Werkzeugs entsprechend einfach. Die grafische Oberfläche lässt sich einfach und intuitiv bedienen. Der Generierungsprozess scheint recht schnell zu sein, aber für den produktiven Einsatz eignet sich das Werkzeug wegen seines frühen Entwicklungsstadiums nicht.
- **Integration:** pmMDA lässt sich weder in eine Entwicklungsumgebung noch in ein Buildwerkzeug integrieren. Entwicklungsumgebungen können das generierte Projekt zwar leicht integrieren, aber es wird keine Unterstützung mittels pmMDA dafür bereitgestellt. pmMDA kann zwar alle Artefakte für das Buildwerkzeug Ant generieren, aber Ant kann damit den Transformations-Prozess nicht anstoßen.

Fazit:

pmMDA ist ein noch sehr unausgereiftes und sich in einer frühen Entwicklungsphase befindendes Werkzeug. Der schlechte Support erschwert zudem den Einstieg. Somit ist das Werkzeug (noch) nicht empfehlenswert.

4.3.6 Fazit Generierungswerkzeuge

Auf Seiten der Generierungswerkzeuge zeigt sich ein sehr viel erfreulicheres Ergebnis, als bei den Modellierungswerkzeugen. Allgemein sind die Werkzeuge ausgereift und können auch produktiv eingesetzt werden. Die meisten Werkzeuge sind allerdings für eine spezielle Anwendungsdomäne optimiert und sollten auch nur in dieser eingesetzt werden.

Eine komplette Umsetzung der MDA erreicht allerdings keines der Werkzeuge. Meist fehlt es an der Modell-zu-Modell Transformation, die auch von keinem der Werkzeuge standardkonform umgesetzt wird. Zudem ist die Interoperabilität auch noch verbesserungswürdig.

4.4 Fazit

In diesem Abschnitt werden die zentralen Ergebnisse und Erkenntnisse, die aus der Evaluierung hervorgegangen sind, aufgeführt.

4.4.1 Modellierungswerkzeuge

Das Generierungswerkzeug gibt im Allgemeinen an, welches Modellierungswerkzeug benutzt werden kann. Fast immer wird hier ArgoUML genannt. Sollte kein Open Source Werkzeug kompatibel sein, sollte besser auf ein kommerzielles Werkzeug zurückgegriffen werden, als zu versuchen, den Generator anzupassen.

4.4.2 Generierungswerkzeuge

Bei den Generierungswerkzeugen hat man eine größere Auswahl. Allerdings sollte man sich seiner Zielplattform genau bewusst sein, denn die unterschiedlichen Werkzeuge eignen sich für einige Plattformen mehr und für andere weniger. Jedes hat seine speziellen Stärken und Schwächen, welches es für eine bestimmte Domäne besonders attraktiv macht.

Bei den Funktionen fällt auf, dass die Modell-zu-Modell Transformation noch so gut wie nicht unterstützt wird. Dies liegt auch an der MOF-QVT Spezifikation der OMG, die noch immer nicht fertig gestellt ist.

4.4.3 Empfehlung

Fast uneingeschränkt lässt sich AndroMDA empfehlen. Es kann in kleinen und großen Projekten problemlos eingesetzt werden und ist für viele Domänen einsetzbar.

Für große Projekte, in denen mit einer DSL oder einem eigenen Metamodell gearbeitet werden soll, ist openArchitectureWare besonders empfehlenswert.

Acceleo ist empfehlenswert, wenn man vor hat, einen kommerziellen Support zu dem Werkzeug zu kaufen. Bei diesem Werkzeug wird ein sehr umfangreicher kommerzieller Support angeboten.

Alle anderen Werkzeuge sind nur bedingt empfehlenswert. Oft ist der Support nicht optimal oder wichtige Funktionen sind unnötig kompliziert.

Für die Zusammenarbeit mit einem der evaluierten Generierungswerkzeuge ist nur ArgoUML empfehlenswert. Besser wäre hier allerdings, zu einem kommerziellen Mo-

dellierungswerkzeug, wie Magic Draw oder Poseidon, zu greifen, da diese deutlich besser unterstützt werden und von der Bedienung weit vor ArgoUML liegen.

Für die Umsetzung des Geschäftsvorfalles werde ich deshalb mit AndroMDA und ArgoUML arbeiten, da diese am besten zu der gegebenen Situation passen. AndroMDA stellt sehr viele Cartridges von Haus aus zur Verfügung, die für die Implementierung des Geschäftsvorfalles verwendet werden können. Ebenso ist der sehr gute Support, der einen schnellen Einstieg in das Werkzeug erleichtert, von großem Vorteil.

5 Prototyp

In diesem Kapitel wird der erstellte Prototyp von der Analyse bis zur Umsetzung beschrieben.

5.1 Analyse

In diesem Abschnitt wird die Analyse der Anforderungen an den Prototypen beschrieben.

5.1.1 Geschäftsvorfall

Ein realistischer Geschäftsvorfall soll als Basis für die Umsetzung des Prototypen dienen.

5.1.1.1 Vorgaben für die Umsetzung des Geschäftsvorfalls

Für die prototypenhafte Umsetzung des Geschäftsvorfalls liegt die Priorität auf der jeweiligen Technologie und weniger auf der fachlichen Korrektheit. Es soll also gezeigt werden, dass mit MDA die Umsetzung des Geschäftsvorfalles technisch möglich ist, auf fachliche Details kann hingegen verzichtet werden. Ebenso sollen die Grenzen der Technologie erforscht werden. Dabei soll gezeigt werden, dass folgende Artefakte aus dem Modell erzeugt werden könne:

- Persistenz
- Fachliche Logik
- Arbeitsabläufe (Workflow)
- Middleware (Kommunikation)

- Benutzer Frontend

Ebenso soll die Erweiterbarkeit des Generierungswerkzeugs aufgezeigt werden:

- Erweiterung von mitgelieferten Transformationen
- Hinzufügen neuer Zielplattformen

Für die fachliche Funktionalität soll ein Geschäftsvorfall aus dem Versicherungsbereich abgebildet werden, der sich auf relativ abstraktem Niveau bewegt. Das ganze soll einfach und überschaubar sein.

Der Geschäftsvorfall ist folgendermaßen vorgegeben und beschreibt einen Vertragsabschluss:

Abbildung 5-1: Business Anwendungsfall für den Geschäftsvorfall

5.1.1.2 Beschreibung des Geschäftsvorfalls

Der Ablauf des Geschäftsvorfalls soll relativ einfach sein, aber trotzdem eine gewisse Komplexität haben. Es gibt zwei Verantwortungsbereiche. Zum einen den des Kunden und zum anderen den des Versicherungssachbearbeiters.

Der Kunde wählt ein entsprechendes Produkt aus, gibt die dafür benötigten Daten ein und sendet den Antrag an die Versicherung. Auf Seiten der Versicherung bearbeitet ein Sachbearbeiter den Antrag. Er prüft dabei die eingegebenen Daten und erstellt anschließend eine Police und eine Rechnung.

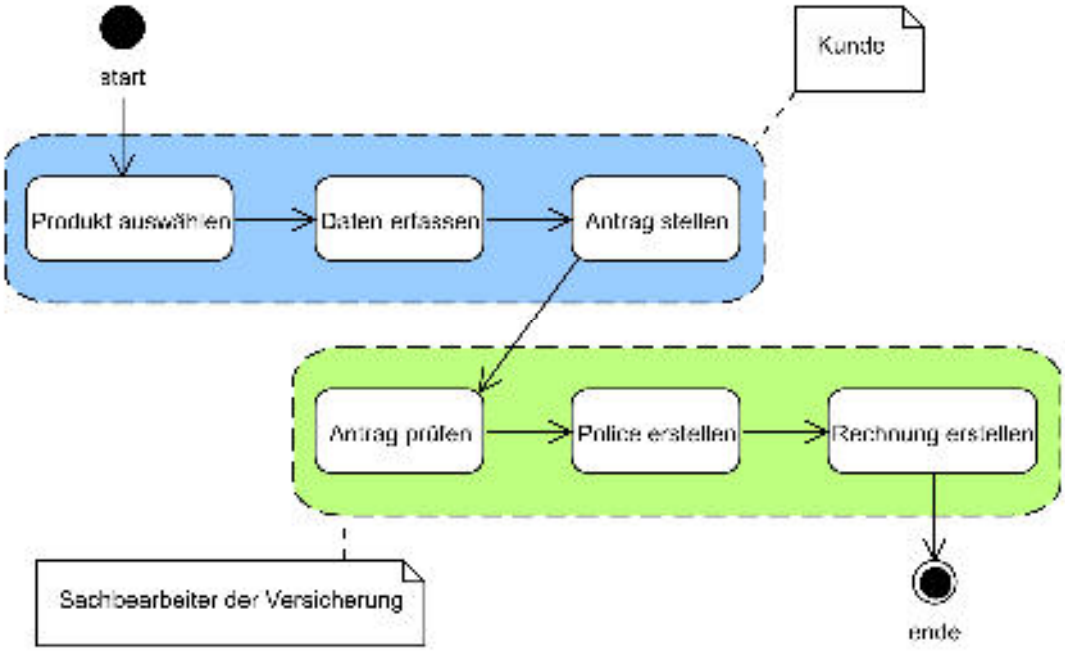

Abbildung 5-2: Ablauf des Geschäftsvorfalls

- **Persistenz** lässt sich an allen benötigten Entitäten, wie beispielsweise *Kunde* oder *Antrag* anwenden und zeigen.
- Entsprechende **Fachlogik** lässt sich beispielsweise in einem Antragsservice umsetzen, der Anträge empfangen und verarbeiten kann.
- Der Übergang der Zuständigkeit vom Kunden zum Sachbearbeiter ist eine passende Stelle, um einen entsprechenden **Prozess oder Workflow** anzustoßen, der den Antrag dann einem geeigneten Sachbearbeiter oder einer Sachbearbeitergruppe zuordnet.
- **Kommunikation zwischen Middleware**-Komponenten findet an vielen Stellen statt, beispielsweise zwischen Persistenzschicht und Geschäftslogikschicht.
- Für das **Benutzer Frontend** bietet sich eine Weboberfläche an, speziell im Zuständigkeitsbereich des Kunden. Der Sachbearbeiter könnte ebenfalls mit einer Weboberfläche arbeiten.
- Für die **Erweiterung mitgelieferter Transformationen** würde sich die Spring Cartridge anbieten. Es könnten zusätzlich Testfälle für modellierte Services generiert werden.
- Für das Hinzufügen neuer Zielplattformen würde sich ein Frontend Client anbieten. Der Sacharbeiter könnte mit einem Fat-Client arbeiten, was ebenfalls realistisch ist.

5.2 Architektur

In diesem Abschnitt wird die Auswahl der Architektur, der verwendeten Technologien und Frameworks beschrieben.

5.2.1 Verwendete Technologien

Da der Geschäftsvorfall eine typische Middleware-Anwendung darstellt, habe ich mich dazu entscheiden, die Anwendung mit J2EE 1.4 und einer klassischen 4-schichtigen Architektur umzusetzen. Die Entscheidung wurde dadurch gestärkt, dass AndroMDA sehr viele Cartridges in diesem Bereich anbietet. Eine Alternative wäre .NET gewesen. Dafür existieren aber leider nicht so viele Cartridges und Erfahrungen im Zusammenhang mit AndroMDA.

Der nächste Entscheidungspunkt liegt bei der Verwendung von EJB oder Spring in Kombination mit Hibernate. Der Vorteil von EJB ist vor allem die lange Erfahrung mit der Technologie und vielen verfügbaren Best Practices. Der Vorteil von Spring sind die leichtgewichtigen Komponenten, basierend auf Plain Old Java Objects (POJOs), die auch keine Einschränkungen in der Modellierung mit sich bringen, wie EJBs, die keine Vererbung kennen. Hibernate bietet außerdem ein flexibleres Objekt/Relationales Mapping, als das mit EJBs möglich ist. Aus diesen Gründen habe ich mich für Spring in Kombination mit Hibernate entschieden. Stände schon EJB3 zur Verfügung, würde die Entscheidung eventuell anders aussehen.

Im Backendbereich habe ich mich für das relationale Open-Source-Datenbankmanagementsystem (RDBMS) MySQL 5.0 entschieden, da ich damit schon viele Erfahrung gemacht habe und in Version 5 viele Funktionen wie Views, Stored Procedures oder Trigger hinzugekommen sind, die die Datenbank auch ernsthaft im Unternehmen einsetzbar macht. Daher ist meine Wahl nicht auf PostgreSQL gefallen.

Für das Frontend standen Struts und JSF zur Auswahl. Es werden für beide Frameworks entsprechende Cartridges von AndroMDA angeboten, wobei die Funktionsvielfalt und Erfahrungen bei der Struts Cartridge deutlich besser sind. Aus diesem Grund habe ich mich für Struts als Frontend-Framework entschieden.

5.2.2 Verwendete AndroMDA Cartridges

Im Folgenden werden alle für die Umsetzung des Geschäftsvorfalls verwendeten AndroMDA Cartridges aufgelistet und dann im Detail beschrieben.

- Java Cartridge
- Spring Cartridge

- Hibernate Cartridge
- BPM4Struts Cartridge
- jBPM Cartridge

5.2.2.1 Java Cartridge

Der Zweck der Java Cartridge ist es, einfache Java Objekte aus statischen Modellen zu generieren. Dabei bildet die Cartridge vor allem eine Basis für andere Cartridges, wie die EJB oder Spring Cartridge, die auf der Java Cartridge aufbauen.

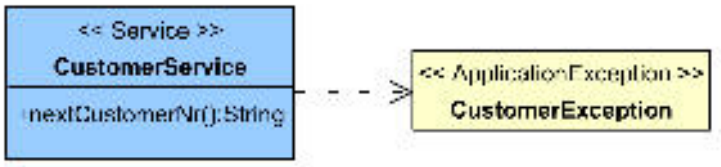

Abbildung 5-3: Beispiel für die Java Cartridge Modellierung

Es werden Stereotypen, unter anderem für Services, Value Objects, Exceptions und Enumerationen, bereitgestellt:

- **Service Stereotyp:** Er wird auf eine Klasse angewendet und es wird entsprechend ein Service-Interface und auch eine Service-Implementation generiert.
- **ValueObject Stereotype:** Er wird auf eine Klasse angewendet und es wird eine Value Object (VO), bzw. Transfer Object (TO) Klasse nach dem gleichnamigen Entwurfsmuster generiert. Dazu muss eine Abhängigkeit von einer Entität zu dem VO gegeben sein.
- **Exception Stereotype:** Er wird auf eine Klasse angewendet und es wird eine entsprechende Java Exception Klasse generiert.
- **Enumeration Stereotype:** Er wird auf eine Klasse angewendet und es wird eine typsichere Java Enumeratiosklasse generiert.

5.2.2.2 Hibernate Cartridge

Die Hibernate Cartridge hat die Aufgabe, die Persistenzschicht für eine Applikation zu generieren.

Es werden Stereotypen, unter anderem für Entitäten, Services und Enumerationen, bereitgestellt:

- **Entity Stereotype:** Er wird auf eine Klasse angewendet und es wird ein entsprechender Hibernate *.hbm.xml Deskriptor (auch Hibernate Mapping genannt) und die dazugehörige Java Klasse generiert.

- **Service Stereotype:** Er wird auf eine Klasse angewendet und es wird im Falle einer Stateful Session Bean der entsprechende Hibernate Deskriptor und die dazugehörige Java Klasse generiert, ähnlich dem Entity-Stereotype.
- **Enumeration Stereotype:** Er wird auf eine Klasse angewendet und es wird eine Typsichere Java Enumeration Klasse generiert. Diese Klasse implementiert das benötigte Hibernate Interface um persistent gemacht zu werden. Dadurch können Entities diesen Enumerationstyp als Attribut benutzen.

Es stehen auch viele Tagged Values zur Verfügung um entsprechende feiner granulare Anpassungen vorzunehmen, beispielsweise kann das Caching-Verhalten sehr detailliert angepasst oder die Hibernate-Vererbungsstrategie verändert werden.

5.2.2.3 Spring Cartridge

Die Cartridge basiert auf dem Spring Framework und kann ohne EJB Container, beispielsweise auf einem standalone Tomcat oder mit EJB Container arbeiten. Die Spring Cartridge funktioniert nur in Zusammenarbeit mit der Hibernate Cartridge.

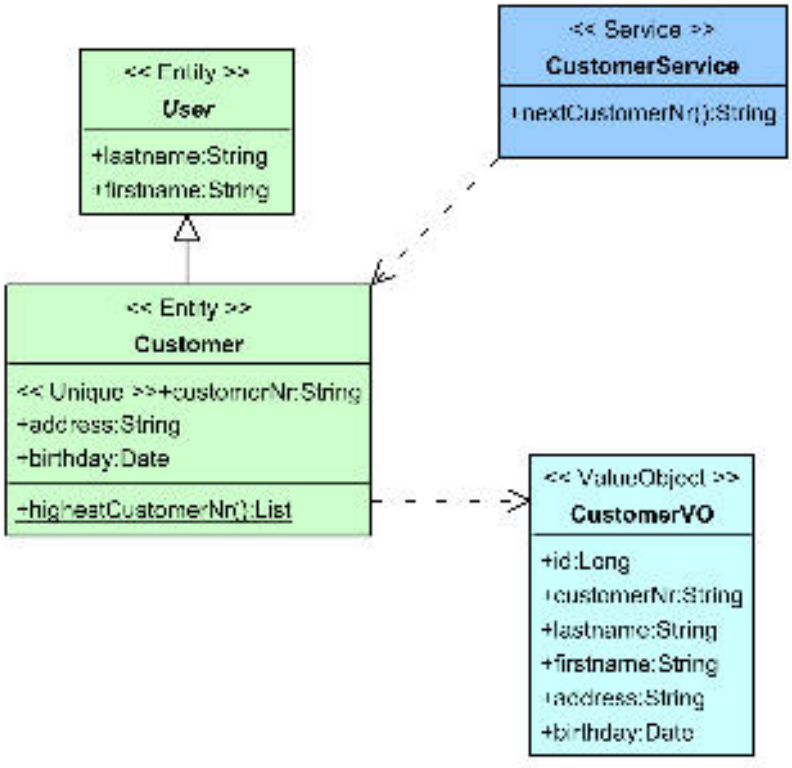

Abbildung 5-4: Beispiel für die Spring und Hibernate Cartridge Modellierung

Es werden Stereotypen, unter anderem für Entitäten, Services und Criterias bereitgestellt:

- **Entity Stereotype:** Er wird auf eine Klasse angewendet und es wird ein Data Access Object (DAO) generiert.

- **Service Stereotype:** Er wird auf eine Klasse angewendet und es wird entweder eine Repräsentation mittels einer EJB Session Bean oder eines Spring Services generiert. Der Service ist als Fassade für mehrere Entitäten gedacht.
- **Criteria Stereotype:** Er wird auf eine Klasse angewendet und es werden mehrere Java Klassen generiert, mit deren Hilfe sich eine Query- oder Finder-Methode einer Entität einschränken lässt.

Die Spring Cartridge bringt ebenfalls einige Tagged Values mit, womit sich Anpassungen, wie beispielsweise das Transaktionsverhalten der Servicemethoden oder das Verhalten eines Criterias, machen lassen.

Es lassen sich auch einfache Abfragen innerhalb einer Entity über OCL definieren, die dann, mittels einer Transformationsbibliothek, in die Hibernate Query Language (HQL) übersetzt werden. Ebenso kann direkt HQL über eine Tagged Value im Modell eingefügt werden, um kompliziertere Abfragen zu erstellen, die mittels OCL nicht machbar sind bzw. unterstützt werden.

Es kann auch das Acegi Security Framework integriert werden. Dann ist es möglich, über die Modellierung von Aktoren den Zugriff auf Services oder Servicemethoden zu sichern. Dabei wird über die Aspektorientierte Programmierung (AOP) des Spring Frameworks ein Security Interceptor eingefügt, der als transparenter Proxy zwischen dem Client und dem Service steht.

5.2.2.4 BPM4Struts Cartridge

Der Zweck der Cartridge ist ein Struts-Web-Frontend aus Aktivitäts- und Anwendungsfalldiagrammen zu generieren. Dabei wird ein Anwendungsfall durch ein Aktivitätsdiagramm näher beschrieben.

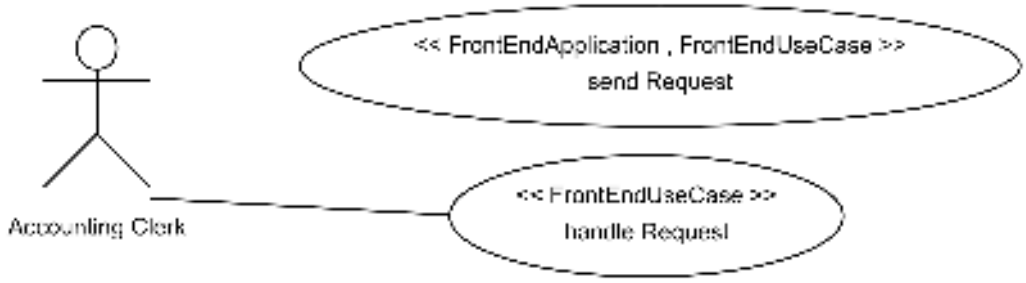

Abbildung 5-5: Beispiel für die Modellierung der Anwendungsfälle für die BPM4Struts Cartridge

Es werden Stereotypen, unter anderem für den Standard Applikationseinstieg, für Session Objekte und für Aktionen, die als Website dargestellt werden sollen, bereitgestellt:

- **FrontEndApplication Stereotype:** Er wird auf einen Anwendungsfall angewendet und markiert diesen Anwendungsfall als Standard-Einstiegspunkt in die Applikation.

- **FrontEndUseCase Stereotype:** Er wird auf einen Anwendungsfall angewendet und markiert diesen als Anwendungsfall für das Struts Framework. Dieser Anwendungsfall muss durch ein Aktivitätsdiagramm näher beschrieben werden.
- **FrontEndView Stereotype:** Er wird auf eine Aktivität angewendet und markiert diese Aktivität als View. Die View wird dann in Form einer JSP generiert.
- **FrontEndSessionObject Stereotype**: Er wird auf eine Klasse angewendet und es wird eine entsprechende Session Klasse generiert, auf die vom Controller aus zugegriffen werden kann, um Daten einfacher über mehrere Seiten hinweg speichern zu können.

Es werden sehr viele Tagged Values bereitgestellt, um die entsprechende Darstellung am Frontend spezifizieren zu können, beispielsweise, um Tabellen darstellen zu können, den Typ eines Formularfeldes zu definieren oder Validationsregeln für Formularfelder zu definieren.

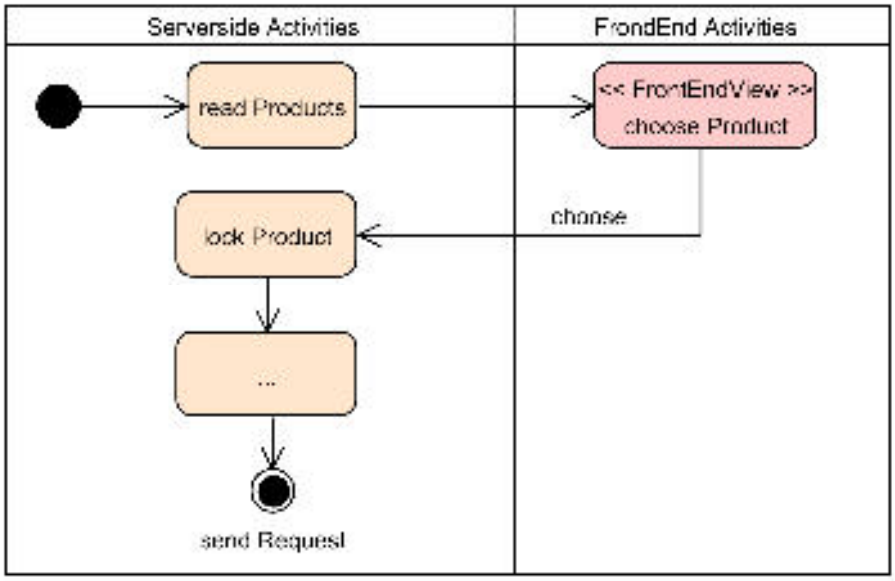

Abbildung 5-6: Beispiel für die nähere Beschreibung des „send Request" Anwendungsfalles durch ein Aktivitätsdiagramm

Über die Modellierung von Aktoren zu den Anwendungsfällen lassen sich Rollen abbilden, die dann über den Java Authentication and Authorization Service (JAAS) überprüft werden können. Somit ist ein weiteres Sicherheitskonzept unterstützt.

5.2.2.5 jBPM Cartridge

Mit der jBPM Cartridge lassen sich Workflows, beziehungsweise Prozesse für die jBPM Workflow-Engine, mittels Aktivitätsdiagrammen modellieren. Die Modellierung ist der, der BPM4Struts Cartridge sehr ähnlich.

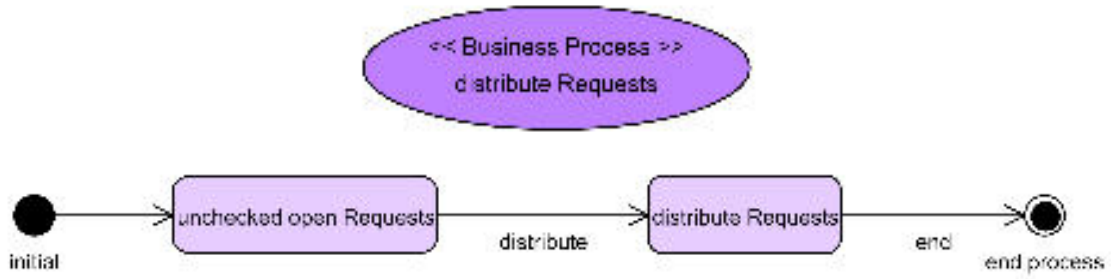

Abbildung 5-7: Beispiel für die jBPM Cartridge Modellierung

Es werden Stereotypen unter anderem für Geschäftsprozesse, Tasks und Timer, bereitgestellt:

- **Business Process Stereotype:** Er wird auf einen Anwendungsfall angewendet und identifiziert diesen als Geschäftsprozess, der durch ein Aktivitätsdiagramm näher beschrieben werden muss.
- **Task Stereotype:** Er wird auf einen Aufruf-Event angewendet. Der Zustand, auf welchen der Event angewendet wird, wird dann zu einem *Task Node* im jBPM-Kontext.
- **Timer Stereotype:** Er wird auf einen Aufruf-Event angewendet. Der Zustand, auf welchen der Event angewendet wird, enthält dann einen Timer, der sich entsprechend regelmäßig wiederholt.

Es werden des Weiteren Tagged Values bereitgestellt, um beispielsweise einen Timer näher zu spezifizieren.

5.2.3 AndroMDAs typische J2EE Architektur

Die durch AndroMDA generierten Anwendungen bauen typischerweise immer auf einer 4-schichtigen Architektur auf:

- **Präsentationsschicht:** Die Präsentationsschicht, auch als Front-End bezeichnet, ist für die Repräsentation der Daten, Benutzereingaben und die Benutzerschnittstelle verantwortlich.
- **Geschäftslogikschicht:** Die Logikschicht beinhaltet alle Verarbeitungsmechanismen. Hier ist sozusagen die Programmintelligenz vereint und mittels festgelegter Schnittstellen kann von der darüber liegenden Schicht darauf zugegriffen werden. Hier laufen unter anderem die Geschäftsprozesse ab.
- **Persistenzschicht:** Die Persistenzschicht regelt die dauerhafte Sicherung der Entitäten und das Objektrelationale Mapping.
- **Datenschicht:** Die Datenschicht enthält die Datenbank und ist verantwortlich für das Speichern und Laden von Daten.

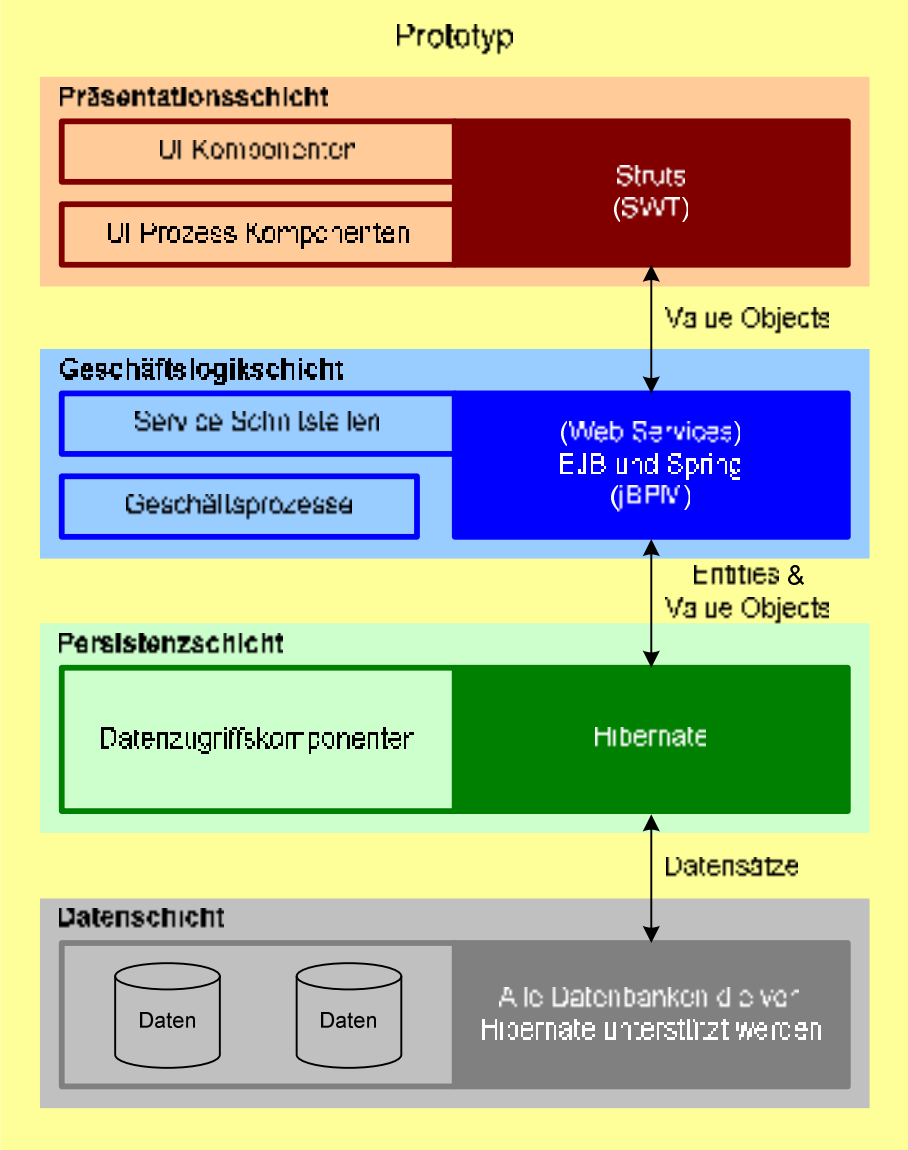

Abbildung 5-8: Die typische 4-Schicht Architektur von AndroMDA J2EE Anwendungen (vgl, [AndroMDA2007], Getting Started – Java)

5.3 Design

In diesem Abschnitt wird das Design des Prototypen beschrieben. Dabei gehe ich hauptsächlich auf die unterschiedlichen Modelle bzw. Konzepte ein.

5.3.1 Domänenmodell

Im Domänenmodell sind die Entitäten und ihre Beziehungen untereinander über ein Klassendiagramm beschrieben.

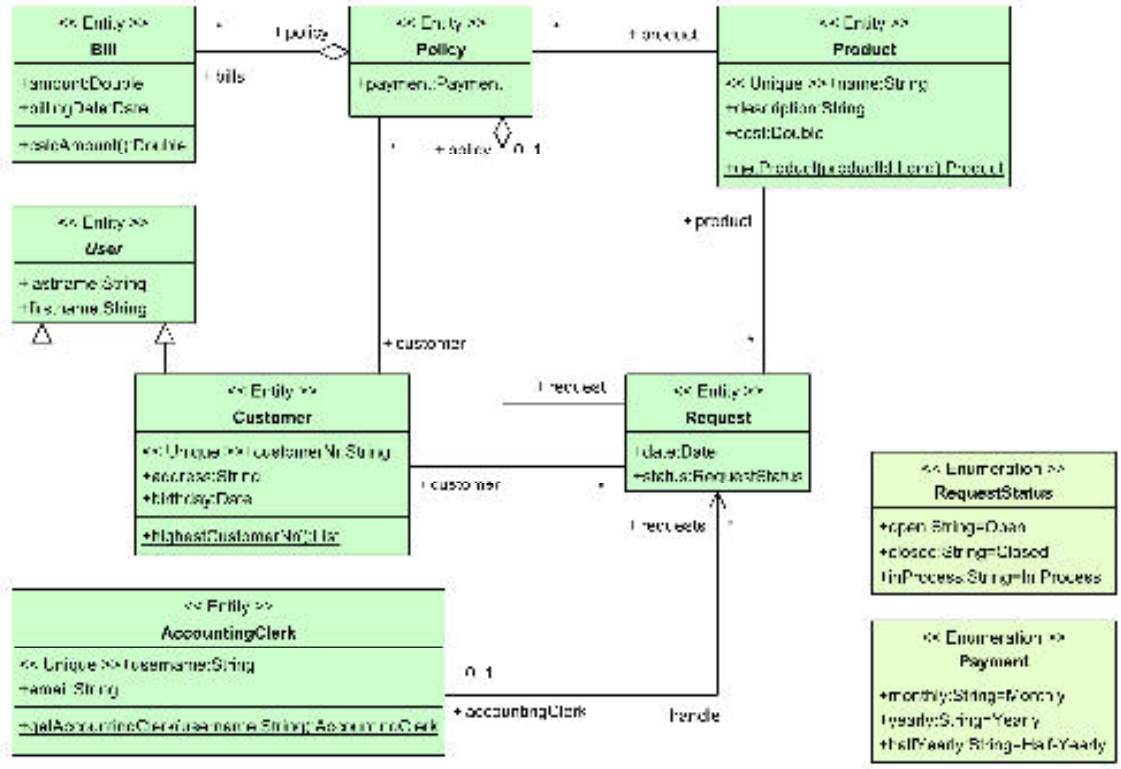

Abbildung 5-9: Domänenmodell des Prototypen

Die Entität **User** stellt einen abstrakten Benutzer des Systems dar. In ihr sind Vor- und Nachname enthalten. Für diese Entität gibt es zwei Spezialisierungen, zum einen die Entität **Customer**, die einen Kunden des Systems abbildet und zum anderen die Entität **AccountingClerk**, die einen Sachbearbeiter der Versicherung abbildet. Die **Request**-Entität fasst quasi Kunde und **Product** zusammen und bildet so einen Antrag ab, der ein Datum und einen Status hat. Der Sachbearbeiter kann diese Anträge nun verwalten und entscheidet, ob aus einem Antrag eine **Policy**-Entität entsteht oder nicht. Eine Policy besitzt entsprechende Rechnungen (**Bill**s), die sich aus dem Produkt und der Policy selbst ergeben. Für den Status eines Antrags und die gewählte Zahlungsweise bei einer Policy, sind entsprechende Enumerationen (**RequestStaus** und **Payment**) vorhanden.

5.3.2 Value Objects

Die Value Objects (oder Transfer Objects) dienen als Austauschobjekte, hauptsächlich zwischen Geschäftslogikschicht und Präsentationsschicht und repräsentieren entsprechende Entitäten mit ihren Attributen. Sie werden üblicherweise nicht per Referenz, sondern als Kopie an den Client übergeben.

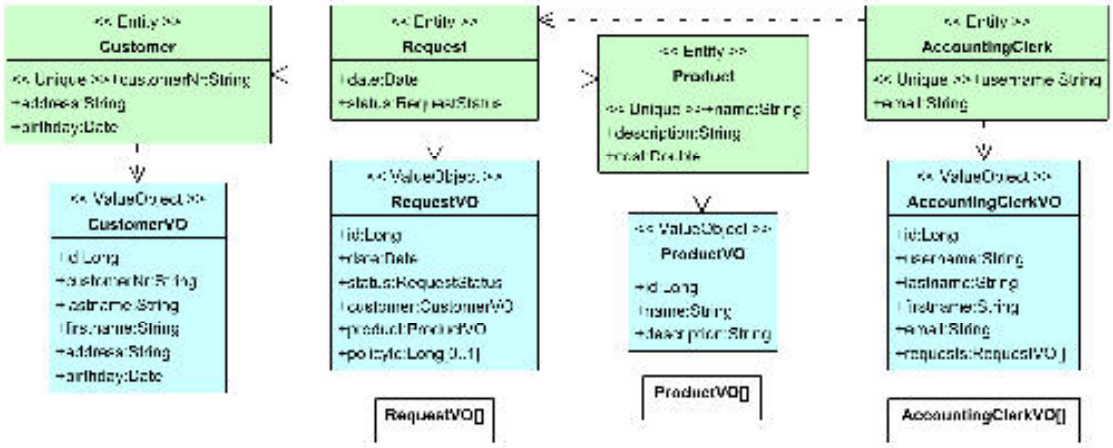

Abbildung 5-10: Auszug aus dem Value Objects Modell des Prototypen

5.3.3 Services

Die Services dienen als Schnittstelle für den Zugriff auf das Domänenmodell und kapseln die Entitäten wie eine Fassade ab. Services sollten möglichst zustandslos sein, dadurch können sie als Singletons Implementiert werden und verbrauchen deutlich weniger Arbeitsspeicher auf dem Applikations-Server und sind dadurch performanter.

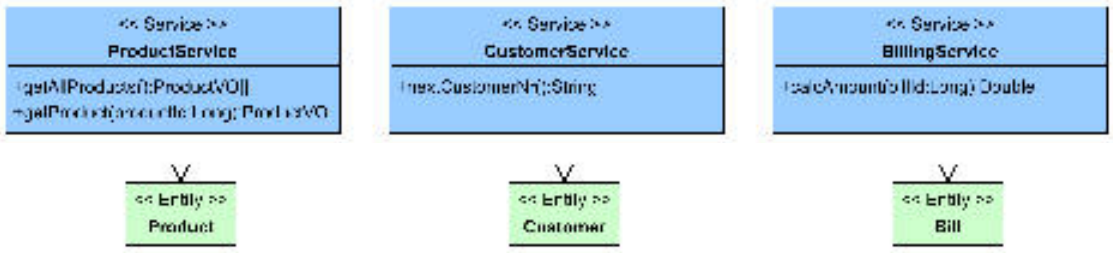

Abbildung 5-11: Auszug aus dem Servicemodell des Prototypen

5.3.4 Benutzer Frontend

Das Benutzer Frontend wird grob über Anwendungsfälle spezifiziert. Der genauere Ablauf wird dann über Aktivitätsdiagramme abgebildet.

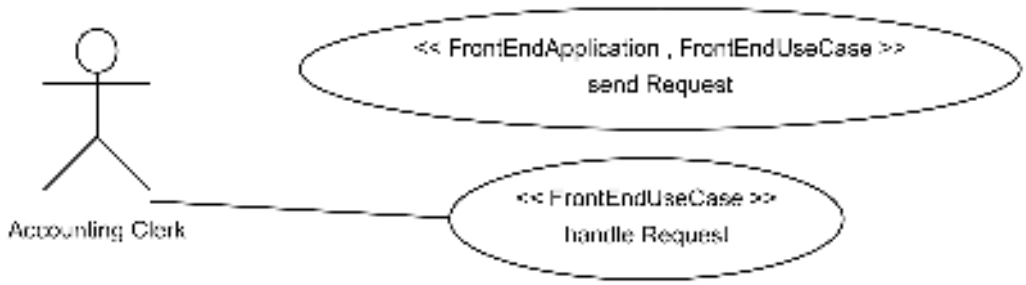

Abbildung 5-12: Das Diagramm beschreibt die Aufteilung der Anwendung in einzelne Anwendungsfälle.

Einem Anwendungsfall ist immer ein Controller zugeordnet, der die Kommunikation zwischen Präsentationsschicht und Geschäftslogikschicht regelt. Zum einfacheren Speichern von Daten während einer Session können Sessionobjekte einem Controller bekannt gemacht werden, die dann entsprechende Daten für eine Session aufnehmen können. Von den Aktivitäten aus kann mittels eines Aufrufereignisses eine Controlleroperation aufgerufen werden.

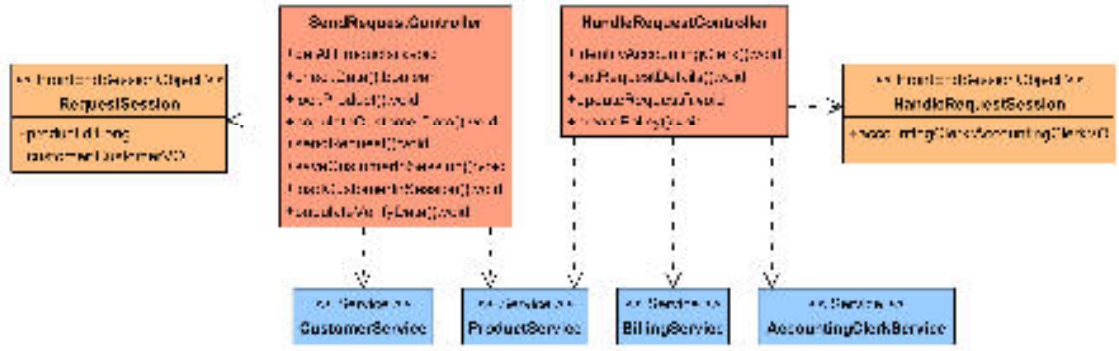

Abbildung 5-13: Modell der Frontend Controller

Der Anwendungsfall „send Request“ startet mit der **read Products** Aktivität. Diese Aktivität ruft über ein Aufrufsereignis die „getAllProducts()“ Methode des Controllers auf. Das Ergebnis wird in der darauffolgenden **choose Products** Aktivität am Frontend angezeigt. Der Benutzer des Frontends kann dann hier sein Produkt auswählen. Hat der Benutzer ein Produkt gewählt, werden die Daten an die **lock Product** Aktivität übergeben, welche dann wieder über ein Aufrufsereignis eine Controllermethode aufruft, welche die Daten dann in dem **RequestSession** Objekt zwischenspeichert.

Der restliche Ablauf läuft nach denselben Mechanismen ab und ist aus den Akivitätsdiagrammen sehr gut nachzuvollziehen. Daher werde ich keine näheren Beschreibungen mehr zu den Aktivitätsdiagrammen der beiden Anwendungsfälle „send Request“ und „handle Request“ machen.

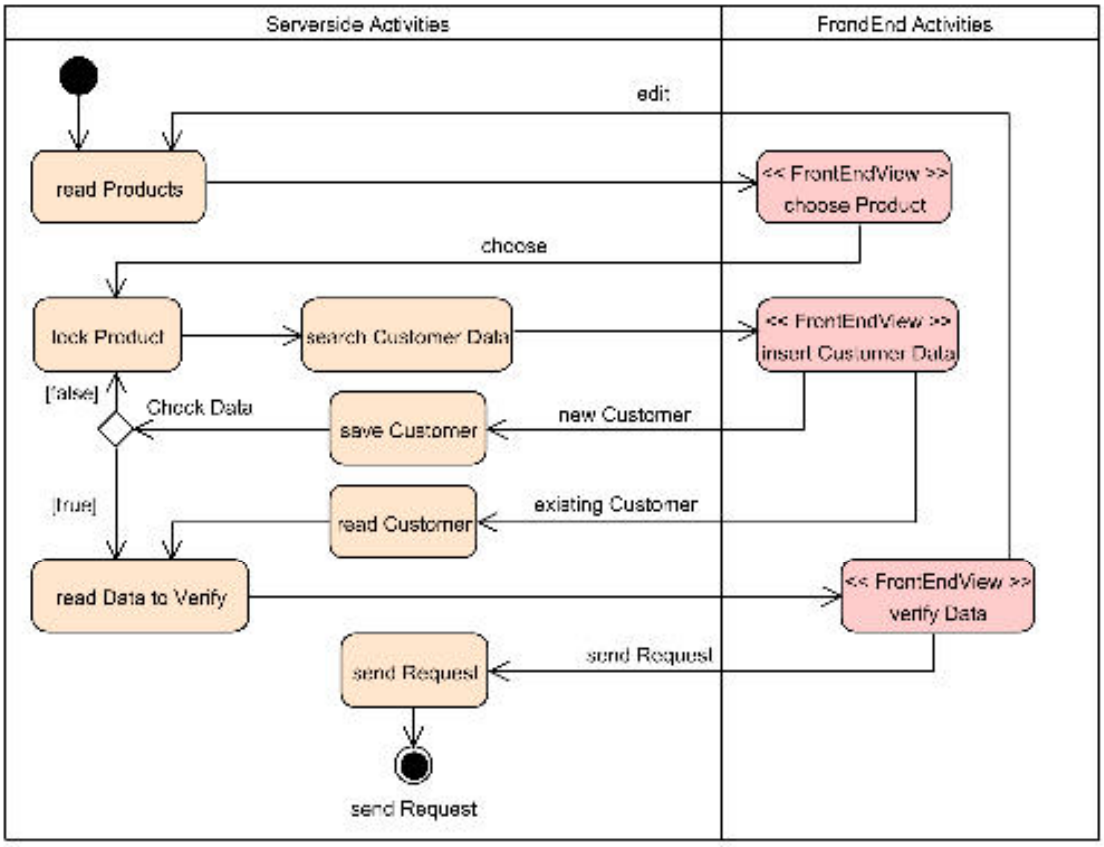

Abbildung 5-14: Das Aktivitätsdiagramm beschreibt den Anwendungsfall „send Request"

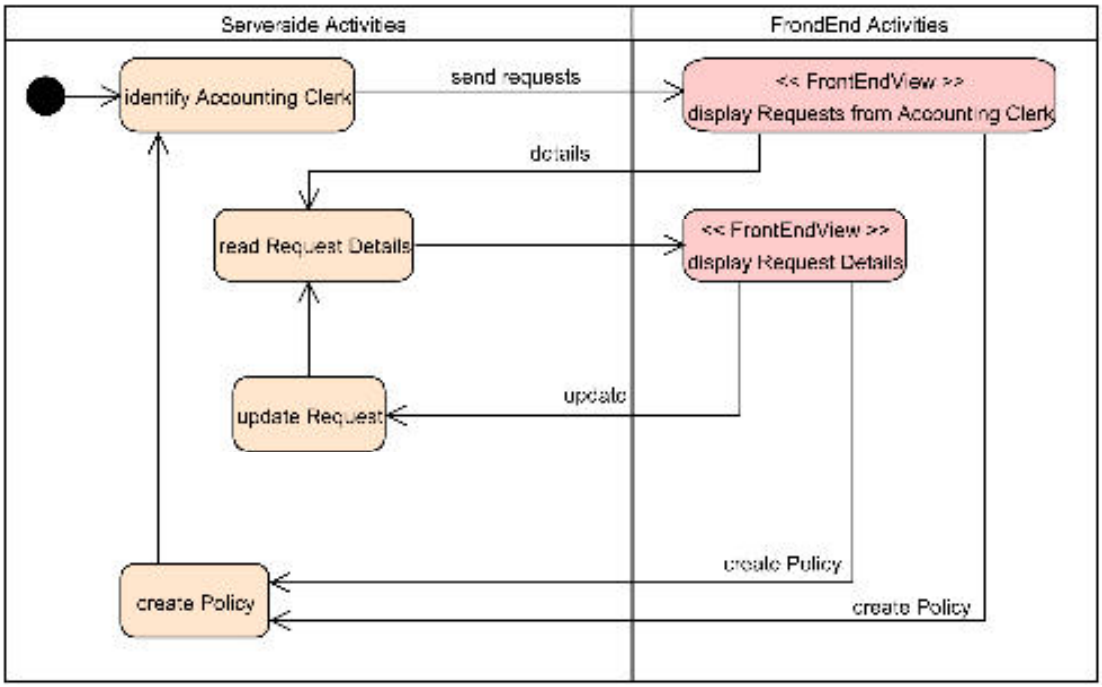

Abbildung 5-15: Das Aktivitätsdiagramm beschreibt den Anwendungsfall „handle Request"

5.4 Umsetzung

In diesem Abschnitt wird die Umsetzung der zuvor aufgeführten Anforderungen aus der Analyse an einem konkreten Beispiel aus dem Prototyp beschrieben. Es kann daher nicht auf alle Details eingegangen werden, somit beschränke ich mich auf die fundamentalsten Eigenschaften. Nähere Details können der AndroMDA Dokumentation [AndroMDA2007] entnommen werden.

5.4.1 Persistenz

Für die Persistenzschicht werden die Entitäten aus dem Domänenmodell verwendet.

Abbildung 5-16: Die Entität Customer (Ausschnitt aus dem Domänenmodell)

Aus dem obigen Modellelement werden mittels der Hibernate Cartridge folgende Artefakte generiert:

- **Customer.hbm.xml:** Die Hibernate Objektrelationale Mapping Datei wird vollständig generiert. Über sie wird dass Mapping zwischen den Objekten und den Tabellen in der Datenbank festgelegt. Normalerweise muss diese Datei nicht weiter editiert werden, da die Mapping Strategie über die zentrale andromda.xml Konfigurationsdatei und / oder über Eigenschaftswerte im Modell festgelegt wird.
- **Customer.java:** Dies ist eine abstrakte Basisklasse, welche vollständig generiert wird und das java.io.Serializable Interface implementiert und ziemlich genau der Klasse aus dem Modell entspricht. Die Klasse hat zusätzlich Getter- und Setter-Methoden zu den modellierten Attributen und enthält eine innere Factory-Klasse, um konkrete *CustomerImpl*-Klassen zu erzeugen.
- **CustomerImpl.java:** Diese Klasse erweitert die *Customer*-Klasse. Wenn normale Methoden modelliert werden (nicht statisch oder query) müssen diese hier Implementiert werden. Außer sie wurden über OCL oder einen Eigenschaftswert vollständig beschrieben, dann sind diese bereits in der *Customer*-Klasse enthalten. Der Rumpf der Klasse und der Methoden wird hierfür generiert.

5.4.2 Fachliche Logik

Fachliche Logik wird mittels Services implementiert. In den Services sollten dann auch die jBPM Prozesse aufgerufen bzw. verarbeitet werden. Ein Service kann auch sehr einfach als Webservice mittels der WebService Cartridge vollautomatisch bereitgestellt werden.

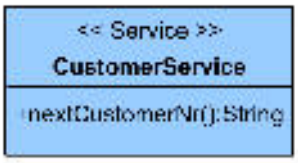

Abbildung 5-17: Der Kunden Service (Ausschnitt aus dem Service Modell)

Aus dem obigen Modellelement werden mittels der Spring Cartridge folgende Artefakte generiert:

- **CustomerService.java:** Dieses Interface, welches vollständig generiert wird, repräsentiert die modellierte *CustomerService*-Klasse.
- **CustomerServiceBase.java:** Diese abstrakte Spring-Basis-Klasse, welche vollständig generiert wird, bietet Zugriff auf alle Services und Entitäten, die von diesem Service aus referenziert werden (*Customer*-Entität).
- **CustomerServiceImpl.java:** Diese Klasse erweitert die *CustomerServiceBase*-Klasse. Hier muss die fachliche Logik des Services implementiert werden. Hierfür wird der Klassen- und Methodenrumpf generiert.
- **ejb/CustomerService.java:** Dieses Interface, welches vollständig generiert wird, erweitert das *javax.ejb.EJBLocalObject*-Interface und ist somit das lokale Interface für die *CustomerServiceBean*-Klasse. Dieses Interface beschreibt die Geschäftsmethoden, die von einem lokalen Client aufgerufen werden können, wie das *CustomerService*-Interface.
- **ejb/CustomerServiceBean.java:** Diese Klasse, welche vollständig generiert wird, erweitert die *org.springframework.ejb.support.AbstractStatelessSessionBean*-Klasse und implementiert das *CustomerService*-Interface. Diese Klasse ist sozusagen identisch mit der *CustomerServiceBase*-Klasse, nur sind hier die vom EJB-Standard geforderten Methoden hinzugefügt.
- **ejb/CustomerServiceHome.java:** Dieses Interface, welches vollständig generiert wird, erweitert das *javax.ejb.EJBLocalHome*-Interface und repräsentiert somit das Home Interface zu der EJB.
- **CustomerServiceException.java:** Diese Klasse ist eine Standard-Ausnahme-Klasse, die automatisch generiert wird, wenn keine Ausnahme-Klasse zu einem Service explizit modelliert wird und erweitert die *ja-*

va.lang.RuntimeException-Klasse. Diese Ausnahme wird von dem Service dann geworfen, falls ein unvorhergesehenes Problem auftritt.

5.4.3 Arbeitsabläufe (Workflow)

Aus dem Modell werden die jBPM-Arbeitsabläufe fast vollständig generiert.

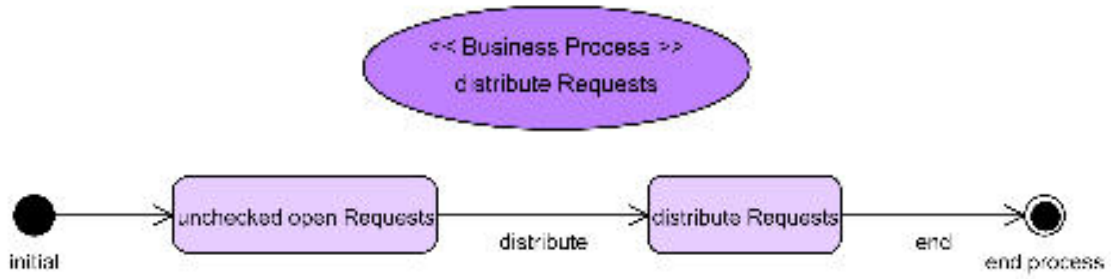

Abbildung 5-18: Das jBPM Modell für die Zuteilung von Anträgen an Sachbearbeiter.

- **distribute-requests.pdl.xml:** Diese Datei ist die zentrale Prozessdefinition, welche vollständig generiert wird. Die Prozessdefinition ist die formale Beschreibung eines Prozesses, welcher mittels Java Objekten repräsentiert wird. In der XML-Datei sind beispielsweise alle Zustände und Knoten mit ihren Beziehungen und Eigenschaften.
- **ProcessManager.java:** Um jBPM in die Anwendung zu integrieren, ist es nötig, die jBPM-Datenbank aufzusetzen und die Prozessdefinition einzufügen. Dafür bietet die *ProcessManager*-Klasse, welche vollständig generiert wird, Methoden an. Wenn dies erledigt ist, kann mit den *Node*-Klassen gearbeitet werden, um die Prozesse zu persistieren.
- **DistributeRequests,java:** Diese Klasse ist eine Helfer-Klasse, welche vollständig generiert wird und statische Methoden bereitstellt, um einfacher die jBPM-Prozess-API für den „distribute Requests“ Prozess zu handhaben.
- **DistributeRequestsProcessNode.java:** Dieses Interface, welches vollständig generiert wird, wird von allen Knoten des „distribute Request“ Prozesses implementiert.
- **DistributeRequestsNode.java:** Diese Klasse, welche vollständig generiert wird, ist die Java Repräsentation des „distribute Requests“ Zustandes.

5.4.4 Middleware (Kommunikation)

Für die einfachere Kommunikation und Kapselung der Persistenzschicht wird ein entsprechendes Data Access Object (DAO) für jede Entität generiert.

Abbildung 5-19: Die Entität Customer (Ausschnitt aus dem Domänenmodell)

Aus dem obigen Modellelement werden mittels der Spring Cartridge folgende Artefakte generiert:

- **CustomerDao.java:** Diese Datei ist ein Java Interface und enthält alle nötigen Methoden, die für die Zugriff auf die Persistenzschicht nötig sind, wie das Laden oder Erzeugen neuer *Customer* Objekte. Dieses Interface wird vollständig generiert.
- **CustomerDaoBase.java:** Diese Klasse ist eine abstrakte Basisklasse, die das *CustomerDao*-Interface implementiert. Sie wird vollständig generiert. Hier sind die meisten Basisfunktionalitäten des DAO implementiert.
- **CustomerDaoImpl.java:** Diese Klasse erweitert die *CustomerDaoBase*-Klasse. Wenn eine statische- oder query-Methode modelliert wurde, so muss diese hier Implementiert werden. Außer sie wurden über OCL oder einen Eigenschaftswert vollständig beschrieben, dann ist diese Methode bereits in der CustomerDaoBase Klasse enthalten. Der Rumpf der Klasse und der Methoden wird hierfür generiert.

5.4.5 Benutzer Frontend

Da für das Benutzer Frontend die Bpm4Struts Cartridge verwendet wird, welche die umfangreichste und komplexeste Cartridge von AndroMDA ist, beschränke ich mich hier auf die zentralen Konzepte, um die Übersichtlichkeit und Verständlichkeit zu wahren. Für nähere Details möchte ich noch einmal auf die AndroMDA Dokumentation [AndroMDA2007] verweisen.

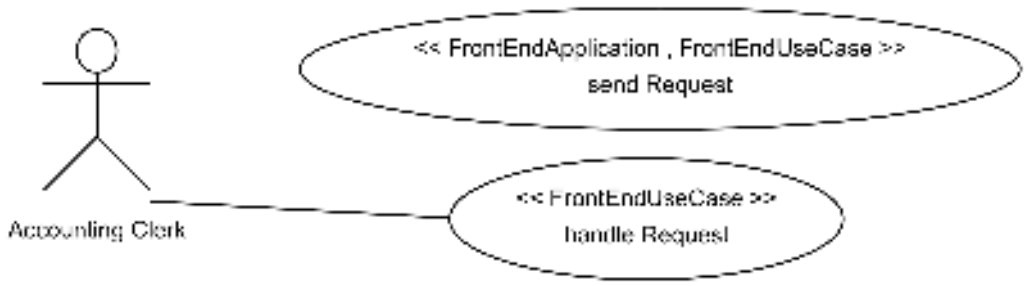

Abbildung 5-20: Die Aufteilung der Anwendung in verschiedene Anwendungsfälle

Für jeden Anwendungsfall wird ein eigenes Paket generiert, das soll die Anwendung Modularer machen. Diese Modularisierung ist beispielsweise auch in der vollständig generierten *struts-config.xml* sichtbar.

Listing 5-1: struts-config.xml – Auszug des „global-forwards"-Teil der zentralen Struts-Konfigurationsdatei

```
…
<global-forwards>
        <forward
            name="send.request"
            path="/SendRequest/SendRequest.do"
            redirect="false" />
        <forward
            name="handle.request"
            path="/HandleRequest/HandleRequest.do"
            redirect="false" />
</global-forwards>
…
```

Für jeden Anwendungsfall muss ein entsprechender Controller existieren. Der Controller stellt die Methoden bereit, die dann im Aktivitätsdiagramm des Anwendungsfalles zur Verfügung stehen.

Abbildung 5-21: Der Controller für den „send Request" Anwendungsfall

Aus dem obigen Modellelement werden mittels der Bpm4Struts Cartridge unter anderem folgende Artefakte generiert:

- **RequestController.java:** Diese Klasse ist eine abstrakte Basisklasse, die das *java.io.Serializable*-Interface implementiert und vollständig generiert wird. Sie enthält abstrakte Methoden der modellierten Methoden, Methoden für das Session-Handling (falls eine Klasse mit dem *FrontEndSessionObject*-Stereotyp verwendet wird), Methoden, um an entsprechend abhängige Services zu gelangen und noch weitere Methoden für Erfolgs- oder Fehlermeldungen.
- **RequestControllerImpl.java:** Diese Klasse erweitert die *RequestController*-Klasse. Es werden die entsprechenden Methodenrümpfe aus dem Modellelement generiert, die Logik der Methoden muss hier manuell implementiert werden.

- **RequestControllerFactory.java:** Über diese Klasse, welche vollständig generiert wird, kommt man an *RequestController*-Instanzen.
- **GetAllProductsForm.java:** Dieses Interface, welches vollständig generiert wird, kapselt alle Felder, die für den Aufruf der *getAllProducts*-Methode des Controllers benötigt werden, ab. Ein solches Interface existiert für jede Methode des Controllers!

Um den Ablauf eines Anwendungsfalles genauer spezifizieren zu können, wird ein Aktivitätsdiagramm verwendet. Dabei ist zu beachten, dass das Aktivitätsdiagramm im Namensraum des Anwendungsfalles und im Kontext des Controllers ist. Dadurch können die Aktivitäten die Methoden des Controllers erst verwenden.

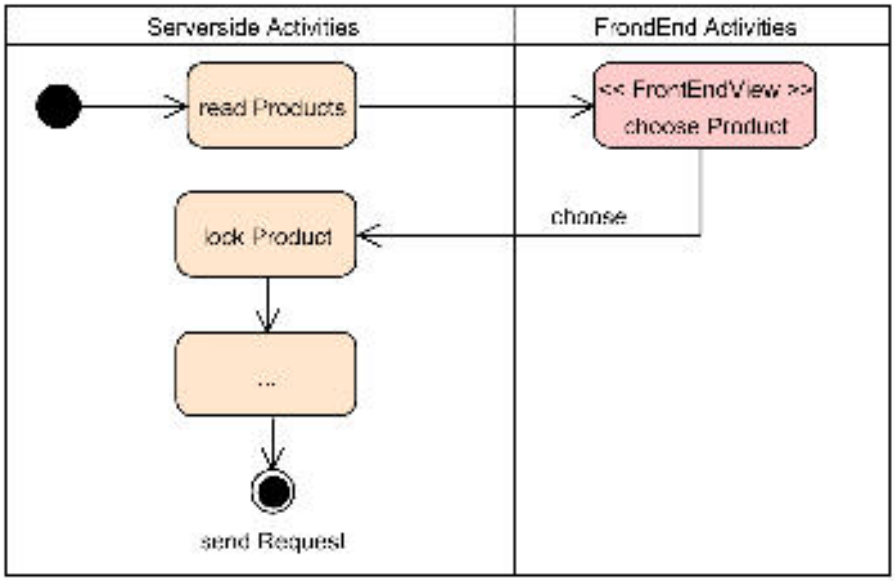

Abbildung 5-22: Ein Ausschnitt aus dem Aktivitätsdiagramm für den „send Request" Anwendungsfall

Aus den obigen Modellelementen werden mittels der Bpm4Struts Cartridge unter anderem folgende Artefakte generiert:

- **ChooseProductChoose.java:** Diese Klasse, welche vollständig generiert wird, erweitert die *org.apache.struts.action.Action*-Klasse. Diese Action-Klasse ist ein Adapter zwischen der eingehenden HTTP-Anfrage und der dazugehörigen Logik im Controller. Hier ist je eine Methode mit einem Aufruf der entsprechenden Controller-Methode für jede serverseitige Aktivität bis zur nächsten *FrontEndView*-Aktivität bzw. bis zum nächsten Endknoten enthalten.
- **ChooseProductChooseFormImpl.java:** Diese Klasse, welche vollständig generiert wird, erweitert die *org.apache.struts.validator.ValidatorForm*-Klasse und Implementiert das *java.io.Serializable* und alle benötigten *<MethodenName>Form*-Interfaces des Controllers. In diesem Fall sind das die Interfaces *LockProductForm* und *…Form*. Diese Klasse bietet eine einfache Feldervalidierung basierend auf einer XML-Datei (*validation.xml*).

- **choose-product.jsp:** Diese JSP, welche vollständig generiert wird, stellt die Struts-Webseite für die „choose Product"-Aktivität im Browser dar.
- **choose-product-choose.jspf:** Diese Datei, welche vollständig generiert wird, enthält den Formularteil für die *choose*-Action und wird in die *choose-product.jsp*-Datei eingebunden. Von hier wird dann die *ChooseProductChoose*-Klasse aufgerufen, wenn der entsprechende Formularknopf im Frontend gedrückt wird.
- **choose-product-javascript.jspf:** Diese Datei, welche vollständig generiert wird, wird ebenfalls in die *choose-product.jsp*-Datei eingebunden und enthält einige Javascripte, die für diese Seite benötigt werden.
- **choose-product.css:** Dieses Cascading Style Sheet (CSS), welches vollständig generiert wird, kann spezielle Formatierungen für die *choose-product.jsp*-Seite enthalten, wo es eingebunden wird.

Abbildung 5-23: Die „choose Product"-Aktivität wie sie am Browser dargestellt wird.

5.4.6 Erweiterung von AndroMDA

5.4.6.1 Erweiterung einer bestehenden Cartridge

Es gibt zwei verschiedene Möglichkeiten, bestehende Cartridges zu erweitern. Beide Möglichkeiten wurden im Prototyp verwendet, um zu den Services entsprechende Testfälle zu generieren.

- **Cartridge Ressourcen überschreiben:** Es lassen sich Ressourcen (Dateien) einer Cartridge überschreiben, ohne diese zu verändern. Dazu muss die „mer-

geLocation“-Eigenschaft in der zentralen AndroMDA Konfigurationsdatei (*andromda.xml*) im jeweiligen Cartridge-Namespace auf einen beliebigen Pfad gesetzt werden. Diese Eigenschaft bewirkt, dass zuerst in diesem spezifizierten Verzeichnis und danach erst in der eigentlichen Cartridge nach Ressourcen für diese Cartridge gesucht wird. Es muss natürlich im spezifizierten Verzeichnis die gleiche Verzeichnisstruktur verwendet werden, wie bei der Cartridge selbst. So kann man beispielsweise ein Template überschreiben.

Listing 5-2: andromda.xml – Setzen der *mergeLocation* Eigenschaft im Spring Namespace

```
…
<namespace name="spring">
      <properties>
            …
            <property name="mergeLocation">
                  ${pom.basedir}/src/customTemplates/andromda-spring
            </property>
…
```

- **Merging von Cartridge Ressourcen:** Es lassen sich auch Daten in eine Ressource einbinden. Dies ist sehr nützlich wenn man keine komplette Ressource ersetzen will und beispielsweise nur eine Zeile einer Konfigurationsdatei in einer Cartridge hinzufügen will. Dazu muss eine Eigenschaft der *andromda.xml* hinzugefügt werden, über die eine Datei spezifiziert wird, welche die Mapping-Informationen enthält. Das Ganze funktioniert nach einem sehr einfachen „Suchen und Ersetzen“-Prinzip. Es werden alle Ressourcen einer Cartridge durchsucht und der Inhalt des *from*-Tags aus einem Mapping wird durch den Inhalt des *to*-Tags ersetzt (siehe Listing unten).

Listing 5-3: andromda.xml – Setzen der mergeMappingsUri im Spring Namespace

```
…
<namespace name="spring">
      <properties>
            …
            <property name="mergeMappingsUri">
                  file:${conf.dir}/mappings/SpringMergeMappings.xml
            </property>
…
```

Listing 5-4: SpringMergeMappings.xml – Definition der Mapping-Regeln

```
<mappings name="SpringMergeMappings">
    <mapping>
        <from>
            <![CDATA[<!-- cartridge-template merge-point -->]]>
        </from>
        <to>
            <path>
                ../../../customTemplates/andromda-spring/META-
INF/andromda/cartridge-template_merge-point.xml
            </path>
        </to>
    </mapping>
    <mapping>
        <from>
            <![CDATA[<!-- cartridge-property merge-point -->]]>
        </from>
        <to>
            <![CDATA[
            <property reference="service-test-impls"/>
            ]]>
        </to>
    </mapping>
</mappings>
```

5.4.6.2 Entwicklung einer Cartridge für eine neue Zielplattform

Hier möchte ich den Entwicklungsprozess für eine AndroMDA Cartridge beschreiben. Dies ist besonders in realen Projekten wichtig, da dort die bereits existierenden Cartridges oft nur bedingt verwendet werden können. Grund hierfür sind komplexe und spezielle Anforderungen der Projekte.

Der Entwicklungsprozess für eine AndroMDA Cartridge besteht im Wesentlichen aus folgenden zehn Schritten:

1. Analysieren der Zieltechnologie (PSM)
2. Identifizieren, designen und generieren der PSM Metaklassen
3. Identifizieren der Transformationsregeln
4. Modellieren, generieren und schreiben von Metafacades
5. Schreiben von Templates
6. Schreiben von Deployment Deskriptoren
7. Erstellen von UML-Profilen
8. Erstellen eines Cartridge-Test-Modells
9. Testen der Cartridge
10. Deployen der Cartridge

Die wichtigsten Schritte möchte ich in den folgenden Abschnitten an einem Beispiel für ein SWT Frontend näher erläutern.

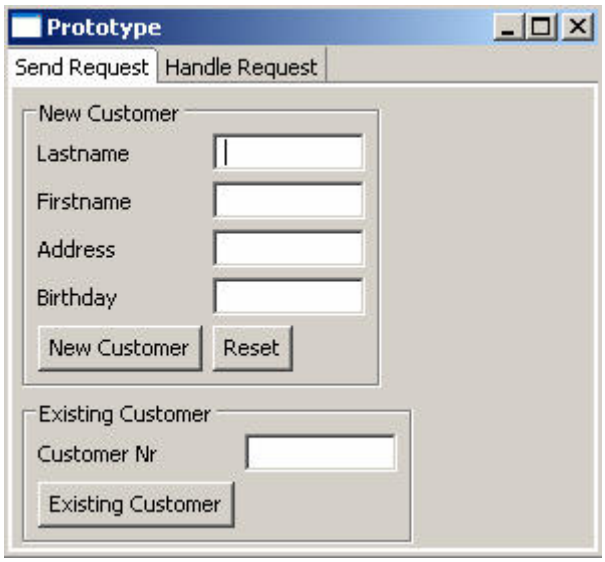

Abbildung 5-24: Ein Prototyp, wie das generierte SWT Frontend aussehen soll.

5.4.6.2.1 Analysieren der Zieltechnologie (PSM)

Als erster Schritt muss die Zieltechnologie analysiert werden, für welche Code generiert werden soll. Dafür sollte man sich einige Fragen stellen:

- Was sind die wesentlichen Artefakte der Zieltechnologie?
    - Java: Klassen, Schnittstellen, Methoden, …
    - Datenbank: Tabellen, Zeilen, Spalten, Schlüssel, …
    - EJB: Entity Beans, Session Beans, Remote Interfaces, …
- Welches Format haben die Artefakte der Zieltechnologie?
    - Java: Source Code
    - Datenbank: DDL Skripte, SQL Skripte mit Test-Daten
    - XML-Schema: XML-Dokumente

Die herausgearbeiteten Artefakte sind dann typische Kandidaten für die PSM-Metaklassen. Das Format der Artefakte bestimmt dann die Umsetzung der Templates.

Im jetzigen Beispiel sind die wesentlichen Artefakte der Technologie folgende:

- **TabItem:** Entspricht einem Tab-Element in einer Tableiste.
- **Composite:** Dient als Container anderer grafischer Elemente.
- **Group:** Gruppiert andere grafischer Elemente.
- **Button:** Entspricht einem Aktionsknopf.
- **Text:** Entspricht einem Textfeld.
- …

Das Format ist in diesem Fall Java Source Code.

5.4.6.2.2 Identifizieren, designen und generieren der PSM Metaklassen

Der nächste Schritt ist aus den Artefakten der Zieltechnologie ein PSM-Metamodell zu erstellen. Dazu müssen PSM-Metaklassen aus den Artefakten abgeleitet werden, dies geschieht oft 1:1.

Innerhalb einer PSM-Metaklasse beschreiben die Attribute den Inhalt der Zielartefakte und die Assoziationen zwischen den Metaklassen beschreiben die logische Struktur der Zieltechnologie.

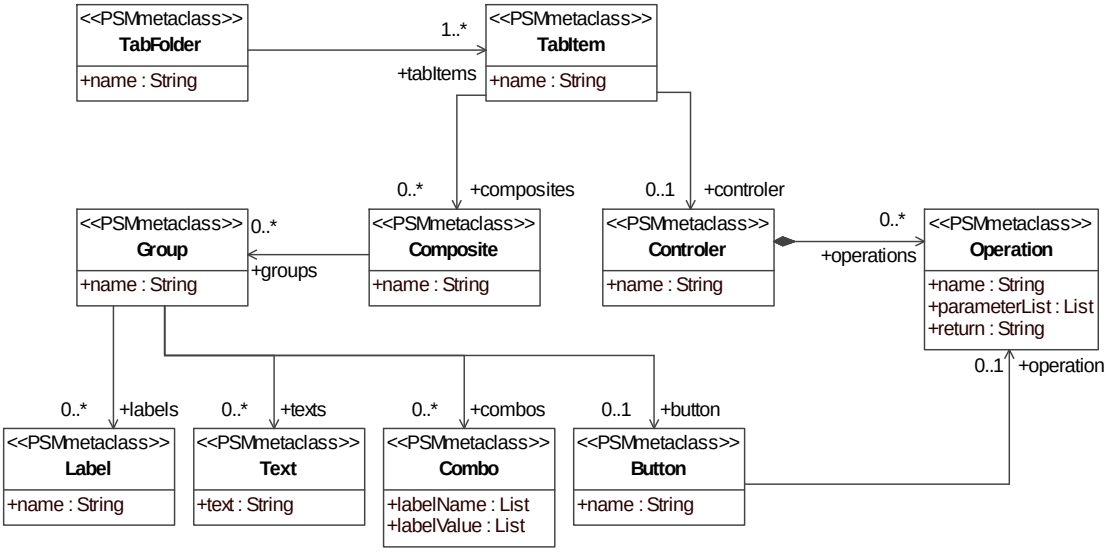

Abbildung 5-25: SWT PSM Metamodell

Hier im Beispiel wird ein ähnliches Verhalten für SWT wie bei der BPM4Struts Cartridge angestrebt.

5.4.6.2.3 Identifizieren der Transformationsregeln

Da jetzt das PSM genau beschrieben ist, müssen noch Transformationsregeln identifiziert werden. Zuerst vom PIM zum PSM und danach vom PSM zum Code.

Dazu erstellt man am besten eine Tabelle:

Tabelle 5-1: Transformationsregeln für das SWT Beispiel

Quellelement im PIM	Zielelemente im PSM	Transformationsbeschreibung
Anwendungsfall	TabFolder	Ein <<FrontEndApplication>> Anwendungsfall erzeugt ein TabFolder.
Anwendungsfall	TabItem	Jeder <<FrontEndUseCase>> Anwendungsfall erzeugt ein TabItem.
Aktivität	Composite	Jede <<FrontEndView>> Aktivität erzeugt ein Composite.
Transition	Group, Button	Jede von einer <<FrontEndView>> Aktivität ausgehende Transition erzeugt eine Gruppe mit einem enthaltenen Button.
Klasse	Controler	Jede Klasse, der ein Aktivitätsdiagramm zugeordnet ist, erzeugt einen Controler.
…	…	…

5.4.6.2.4 Modellieren, generieren und schreiben von Metafassaden

Da die PIM-Metaklassen im AndroMDA-Kontext immer durch Metafassaden abgeschirmt werden, müssen diese noch modelliert und anschließend implementiert werden. Die vorher erarbeiteten Transformationsregeln werden als Operationen in den Metafassaden umgesetzt, die dann entsprechende PSM Metaobjekte zurückgeben. So wird mit AndroMDA eine PIM-zu-PSM Transformation implementiert.

Anhand der im vorigen Schritt identifizierten PIM Quellelemente sucht man sich entweder schon von AndroMDA passende mitgelieferte Metafassaden oder erstellt eine Neue. Erstellt man eine Neue, so muss man darauf achten, dass eine Abhängigkeit zu der entsprechenden PIM Metaklasse modelliert wird, welche die Metafassade abschirmen soll.

Anschließend spezialisiert man die Metafassaden und fügt den spezialisierten Metafassaden entsprechende Operationen hinzu, die dann die PIM-zu-PSM Transformationen durchführen.

Als nächstes benutzt man die AndroMDA Meta- und Java-Cartridge, um sich die entsprechenden Klassen zu generieren.

Nun müssen die Operationen der Metafassaden für die PIM-zu-PSM Transformation noch durch entsprechende Algorithmen Implementiert werden.

5.4.6.2.5 Schreiben von Templates

Als nächstes müssen Templates geschrieben werden, die die PSM-zu-Code Transformation durchführen. Dabei können die Templates nur auf die Metafassaden zugreifen und erhalten durch die Transformationsoperationen dieser die PSM Metaobjekte. Anhand der PSM Metaobjekte wird dann die Transformation zum Code durchgeführt.

5.5 Fazit

Durch die Implementierung des Prototypen hat sich gezeigt, dass durch den Einsatz von MDA ein großer Teil der Anwendung generativ erstellt werden kann. Die betrachteten Anforderungen konnten fast alle erfüllt bzw. gezeigt werden. Nur bei den Workflows und bei der Entwicklung einer Cartridge für eine neue Zieltechnologie gab es technische bzw. zeitliche Probleme. Generell sind diese Probleme aber lösbar und in Abschnitt 6.1.1 wird näher auf diese Probleme eingegangen.

Lediglich der Ablauf der Prozesse, bzw. die Logik der Anwendung, muss weiterhin manuell implementiert werden. Die Verlagerung dieser Logik in die Modelle ergibt zum heutigen Zeitpunkt, aus meiner Sicht, keinen Sinn, da die verfügbaren Mittel und Werkzeuge hierfür noch unzureichend sind. Es würde außerdem der Vorteil der Modelle,

dass diese einfach und übersichtlich sind, verloren gehen. Hier ist also die momentane Grenze der MDA.

6 Zusammenfassung

In diesem Kapitel wird die Arbeit noch einmal zusammengefasst. Es wird auf Probleme und Ausblicke eingegangen.

6.1 Ergebnisbewertung

In diesem Abschnitt wird auf Probleme während der Diplomarbeit eingegangen und der Einsatz von MDA im Unternehmen diskutiert.

6.1.1 Probleme

Während der Diplomarbeit gab es natürlich auch Hürden, die genommen oder umgangen werden mussten. Die beiden Wichtigsten will ich an dieser Stelle näher beschreiben.

- Bei der Anforderung Workflow bzw. Prozesse, wofür AndroMDA die jBPM Cartridge bereitstellt, gab es leider technische Probleme mit der Cartridge während der Prototypenentwicklung. Es hat sich herausgestellt, dass wichtige Konfigurationsdateien (*ehcache.xml, hibernate.cfg.xml, hibernate.properties, jbpm.properties*) nicht im Enterprise Application Archive (EAR) bzw. in einem Java Archive (JAR) innerhalb des EAR enthalten sind. Diese Dateien enthalten unter anderem die Parameter für den Datenbankzugriff. Leider wusste auch niemand im AndroMDA Forum Rat. Somit konnte im Prototypen nicht mit dem Workflow gearbeitet werden.
- Während der Entwicklung der Cartridge für eine neue Zielplattform (SWT Cartridge) hat sich herausgestellt, dass die hierfür notwendige Dokumentation unvollständig ist. Es wird zwar beschrieben, wie alles modelliert und generiert werden soll, aber wie die AndroMDA-Umgebung zu konfigurieren ist, um mit Hilfe der Me-

ta Cartridge Code zu generieren oder wie wichtige Deployment Deskriptoren geschrieben werden müssen, wird nicht behandelt. Außerdem fehlt, wie die zu entwickelnde Cartridge getestet werden kann oder wie das Deployment der Cartridge vonstatten geht. Diese Themen sind zwar in der Dokumentation vorgesehen, aber leider noch ohne Inhalt. Leider konnten auch die vorhandenen Cartridges nicht als Beispiel oder Vorlage herangezogen werden. Diese sind nach einer älteren Methode entwickelt worden und sind nicht an das neue System, wie es auch in der Dokumentation beschrieben ist, angepasst worden. Daher hatte ich zwei Möglichkeiten, zum einen zu versuchen, die Cartridge nach der alten Methode zu entwickeln und dabei eine vorhandene Cartridge als Vorlage zu verwenden. Oder zum anderen, die Entwicklung der Cartridge an diesem Punkt abzubrechen. Ich habe mich dann für die zweite Möglichkeit entschieden und habe die Entwicklung an diesem Punkt abgebrochen. Das Risiko für die Entwicklung nach der älteren Methode war mir zu hoch, da ich dazu nur die Cartridge als Vorlage gehabt hätte und keine Dokumentation. Ich hätte hierbei sehr oft nach der „Trial and Error"-Methode arbeiten müssen und das daraus resultierende Risiko, ebenfalls zu scheitern oder zu viel Zeit zu verbrauchen, war zu hoch. Im Forum ist dieses Problem bekannt, allerdings gab es keine Antworten, bis wann die Dokumentation vervollständigt wird.

6.1.2 MDA im Unternehmen

Ein Ziel der Diplomarbeit ist herauszufinden, ob MDA, mit Open-Source-Werkzeugen, im Unternehmen produktiv eingesetzt werden kann. Diese Fragestellung kann eindeutig mit „ja" beantwortet werden. Allerdings muss im Einzellfall entschieden werden, ob man zwingend ein Open-Source-Werkzeug einsetzen will, oder ob man alternativ ein kommerzielles Produkt wählt. Beispielsweise ist es meiner Meinung nach derzeit in vielen Fällen sinnvoller, ein kommerzielles Modellierungswerkzeug einzusetzen. Siehe dazu auch Abschnitt 4.2.6.

6.2 Stand der Technik und Ausblick

Der aktuelle Stand der Technik ist soweit, das MDA bereit für den produktiven Einsatz im Unternehmen ist. Die Zukunft von MDA erscheint vielversprechend, allerdings muss dafür noch einiges getan werden.

- **Interoperabilität:** An der Interoperabilität zwischen Modellierungs- und Generierungswerkzeug muss noch gearbeitet werden, allerdings scheint es als ob der XMI-Standard langsam einen stabilen Status erreicht und die kommenden Versionen in immer längeren Zeitabschnitten erscheinen werden. Momentan

sind leider noch zu viele verschiedene XMI-Versionen am Markt vertreten. Dies fällt besonders bei den Modellierungswerkzeugen auf. Hier hinken die Open-Source-Werkzeuge den kommerziellen Produkten noch ein Stück hinterher. Aber auch die Interoperabilität unter den Generierungswerkzeugen sollte verbessert werden. Wobei hier bei den Open-Source-Vertretern kaum Unterschiede im Vergleich zu den kommerziellen Produkten auszumachen sind. In ferner Zukunft sollten die Transformationsbeschreibungen, egal ob Modell-zu-Modell oder Modell-zu-Code, unabhängig vom Generierungswerkzeug werden. Dann könnte man sich aber die Frage stellen, in wie weit sich die Generatoren überhaupt noch voneinander unterscheiden. Die Generatoren könnten sich dann ähnlich wie die Applikation-Server entwickeln, die im Prinzip auch alle dieselben Fähigkeiten haben und sich nur an eine Spezifikation halten. Unterscheidungsmerkmale liegen dann natürlich im gebotenen Komfort und der jeweiligen Unterstützung bei den einzelnen Disziplinen.

- **Cartridges:** Bei den Cartridges existieren bei den verschiedenen Werkzeugen sehr unterschiedliche Philosophien. Bei den Meisten gehören Cartridges allerdings zum Werkzeug dazu und decken meist die wichtigsten Plattformen ab. Für die Cartridges könnte sich zukünftig ein Markt eröffnen, an dem ein Unternehmen Cartridges für spezielle Plattformen kaufen kann. Ebenso könnten größere Unternehmen ihre Architekturen über die Cartridges beschreiben. Dadurch hätten dann alle Anwendungen denselben Stil. Dies würde die Wartbarkeit und den laufenden Betrieb verbessern und auch die Weiterentwicklung würde sich einheitlicher gestalten.

- **Modellierung:** Ich denke, die Modellierung von Geschäftslogik wird sich nicht durchsetzen, da hierfür wieder entsprechende textuelle Sprachen, wie OCL, notwendig sind. An diesem Punkt hat die grafische Modellierung ihre Grenzen erreicht, zumindest zum jetzigen Zeitpunkt. Für sehr einfache Konstrukte sind diese textuellen Sprachen sicherlich sinnvoll, aber komplexere Sachverhalte abzubilden steht im Gegensatz zum eigentlichen Sinn der Modellierung, nämlich der Einfachheit und Verständlichkeit. Geschäftsprozesse werden sicherlich auch weiterhin modelliert werden, wie das heutzutage auch schon getan wird.

- **Transformationen:** Fast alle Werkzeuge bieten noch keine Modell-zu-Modell Transformationen an. Fast immer wird direkt vom PIM zum Code transformiert. Allerdings ist zum jetzigen Zeitpunkt erkennbar, dass die Modell-zu-Modell Transformation immer wichtiger werden wird. AndroMDA wird Modell-zu-Modell Transformation in der kommenden Version unterstützen und openArchitectureWare beherrscht seit der jetzigen Version Modell-zu-Modell Transformationen. Die zum heuten Zeitpunkt in den Werkzeugen verwendeten Transformationsmechanismen müssen weiterentwickelt werden. Dabei ist, denke ich, sehr wichtig, dass die Transformationen bidirektional ablaufen können. Also vom Abstrakten zum Konkreteren und umgekehrt, natürlich mit ein

und derselben Transformationsregel. Das hätte den Vorteil, dass die Konsistenz zwischen den unterschiedlich abstrakten Modellen besser gewahrt werden kann.

Ich bin der Meinung, dass sich MDA, oder allgemein die modellgetriebene Entwicklung, mit der Zeit immer weiter durchsetzen und verbreiten wird. Dabei wird sich MDA sicher nicht wie in der reinen Lehre durchsetzen lassen, sondern es werden praktische Erfahrungen aus der allgemeinen modellgetriebenen Entwicklung einfließen. Dabei wird sicherlich auch die weitere Entwicklung der Eclipse-basierten Modellierungskonzepte wie EMF oder GMF eine wichtige Rolle spielen. Eclipse ist in seiner bisherigen Vergangenheit einen sehr praxisnahen Weg gegangen und war damit sehr erfolgreich.

Die generativ erstellten Artefakte sind Heute nicht mehr nur ein Skelett, wie das noch vor einigen Jahren war, sondern sie entsprechen eher einem Skelett mit Muskeln, Organen und einer überspannenden Haut. Nur die Funktionsweise der Organe muss noch manuell hinzugefügt werden. Versucht man allerdings auch die Funktionsweise dieser Organe zu generieren, beziehungsweise zu modellieren, stößt man schnell an die Grenzen von Kosten und Nutzen. Grundsätzlich lässt sich jede Funktionsweise über ein entsprechendes Metamodell darstellen, aber die Kosten übersteigen den davon gewonnenen Nutzen deutlich. Mann muss sich dafür nur eine grafische Programmiersprache vorstellen, mit der man Algorithmen modelliert. Die Kosten übersteigen dabei den Nutzen bei Weitem, da man sicherlich schneller ist, wenn man textuell arbeitet. Ebenso geht die Übersichtlichkeit schnell verloren. Wer schon einmal mit einem grafischen XML-Schema- oder XSLT-Editor gearbeitet hat kann das eventuell nachvollziehen.

Abbildungsverzeichnis

Abkürzungsverzeichnis

MDA	Model Driven Architectur
XMI	XML Metadata Interchange
OMG	Object Management Group
UML	Unified Modeling Language
MOF	Meta-Object Facility
CIM	Computation Independent Model
PIM	Platform Independent Model
PSM	Platform Specific Model
CASE	Computer-Aided Software Engineering
XML	Extensible Markup Language
CWM	Common Warehouse Metamodel
OCL	Object Constraint Language
DTD	Dokumenttypdefinition
xUML	Executable UML
CORBA	Common Object Request Broker Architecture
EJB	Enterprise Java Bean
JMI	Java Metadata Interface
QVT	Queries Views Transformations
JEE	Java Platform, Enterprise Edition
J2EE	Java 2 Platform Enterprise Edition
IDE	Integrated Development Environment
EMF	Eclipse Modeling Framework
KDE	K Desktop Environment
IDL	Interface Description Language
FAQ	Frequently Asked Questions
GPL	General Public License
LGPL	Lesser General Public License
BSD	Berkeley Software Distribution
EPL	Eclipse Public License

GNU	GNU's Not Unix
CPL	Common Public License
PHP	PHP: Hypertext Preprocessor
MDSD	Model Driven Software Development
CRUD	Create Retrieve Update Delete
GMT	Generative Modeling Tools
DSL	Domain-Specific Language
JSP	Java Server Pages
JDBC	Java Database Connectivity
JET	Java Emitter Templates
JAG	Java Application Generator
POJO	Plain Old Java Object
RDBMS	Relational Database Management System
SQL	Structured Query Language
DAO	Data Access Object
VO	Value Object
TO	Transfer Object
HQL	Hibernate Query Language
GMF	Graphical Modeling Framework
XSLT	XSL Transformations
XSL	Extensible Stylesheet Language
DDL	Data Definition Language
CSS	Cascading Style Sheet
EAR	Enterprise Application Archive
JAR	Java Archive

Tabellenverzeichnis

Listingverzeichnis

Literaturverzeichnis

[OMG2003] Jishnu Mukerji, Joaquin Miller; Object Management Group (OMG): MDA Guide Version 1.0.1, 2003, http://www.omg.org/docs/omg/03-06-01.pdf (02.08.2006).

[Hitz2005] M. Hitz, G. Kappel, E. Kapsammer, W. Retschitzegger: UML@Work - Objektorientierte Modellierung mit UML2, 3., aktualisierte und überarbeitete Auflage. dpunkt.verlag, Heidelberg, 2005

[OMG2006] Object Management Group (OMG): OMG Model Driven Architecture, 2006, http://www.omg.org/mda/ (03.08.2006).

[Wikipedia2006] Wikipedia – Die freie Enzyklopädie: Model Driven Architecture, 2006, http://de.wikipedia.org/wiki/Model_Driven_Architecture (07.08.2006)

[Wikipedia2006a] Wikipedia – Die freie Enzyklopädie: Open Source, 2006, http://de.wikipedia.org/wiki/Open_source (29.08.2006)

[GNU2006] GNU's Not Unix: GNU General Public License, 2006, http://www.gnu.org/licenses/gpl.html (29.08.2006)

[GNU2006a] GNU's Not Unix: GNU General Public License, 2006, http://www.gnu.org/licenses/lgpl.html (29.08.2006)

[Opensource2006] Opensource: The BSD License, 2006, http://www.opensource.org/licenses/bsd-license.php (29.08.2006)

[Eclipse2006] Eclipse: Eclipse Public License – v 1.0, 2006, http://www.eclipse.org/org/documents/epl-v10.php (29.08.2006)

[oAW2006] openArchitectureWare (oAW): openArchitectureWare Overview, 2006, http://www.eclipse.org/gmt/oaw/diagram.php (07.09.2006)

[openMDX2006] openMDX: openMDX Introduction, 2006, http://www.openmdx.org/documents/introduction/openMDX_Introduction.pdf (11.09.2006)

[AndroMDA2007] AndroMDA: AndroMDA 3.2 Dokumentation, 2007, http://galaxy.andromda.org/docs/ (10.01.2007)

Anhang A – CD

Inhalt der beigelegten CD:

- **Diplomarbeit:** Diese Diplomarbeit als Druckversion (300dpi) und Bildschirmversion (72dpi).
- **Prototyp:** Der entwickelte Prototyp in Quelltext-Format und als deploybares ear-Archiv. Außerdem sind noch Screenshots der Benutzeroberfläche des Prototypen enthalten.